e-Chapter 7

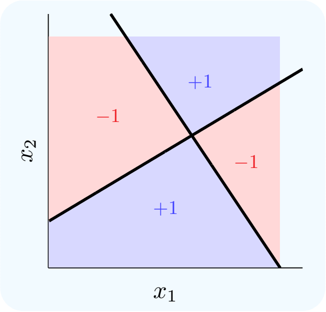
Neural Networks

Neural networks are a biologically inspired[1] model which has had considerable engineering success in applications ranging from time series prediction to vision. Neural networks are a generalization of the perceptron which uses a feature transform that is *learned* from the data. As such, they are a very powerful and flexible model.

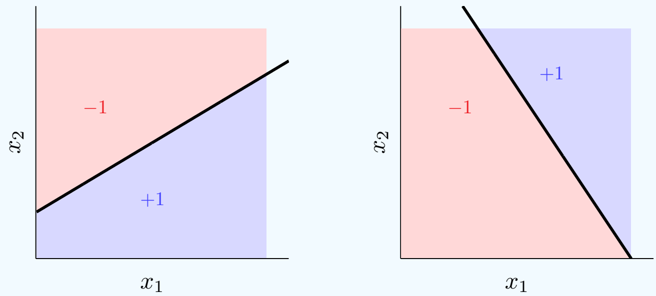
Neural networks are a good candidate model for learning from data because they can efficiently approximate complex target functions and they come with good algorithms for fitting the data. We begin with the basic properties of neural networks, and how to train them on data. We will introduce a variety of useful techniques for fitting the data by minimizing the in-sample error. Because neural networks are a very flexible model, with great approximation power, it is easy to overfit the data; we will study a number of techniques to control overfitting specific to neural networks.

7.1 The Multi-layer Perceptron (MLP)

The perceptron cannot implement simple classification functions. To illustrate, we use the target on the right, which is related to the Boolean XOR function. In this example, f cannot be written as $\text{sign}(\mathbf{w}^{\text{T}}\mathbf{x})$. However, f is composed of two linear parts. Indeed, as we will soon see, we can decompose f into two simple perceptrons, corresponding to the lines in the figure, and then combine the outputs of these two perceptrons in a simple way to get back f. The two perceptrons are shown next.



[1]The analogy with biological neurons though inspiring should not be taken too far; after all, we build planes with wings that do not flap. In much the same way, neural networks, when applied to learning from data, do not much resemble their biological counterparts.



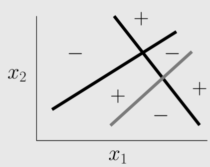
$h_1(\mathbf{x}) = \text{sign}(\mathbf{w}_1^{\mathrm{T}}\mathbf{x})$ $\qquad$ $h_2(\mathbf{x}) = \text{sign}(\mathbf{w}_2^{\mathrm{T}}\mathbf{x})$

The target f equals $+1$ when exactly one of h_1, h_2 equals $+1$. This is the Boolean XOR function: $f = \text{XOR}(h_1, h_2)$, where $+1$ represents "TRUE" and -1 represents "FALSE". We can rewrite f using the simpler OR and AND operations: $\text{OR}(h_1, h_2) = +1$ if at least one of h_1, h_2 equal $+1$ and $\text{AND}(h_1, h_2) = +1$ if both h_1, h_2 equal $+1$. Using standard Boolean notation (multiplication for AND, addition for OR, and overbar for negation),

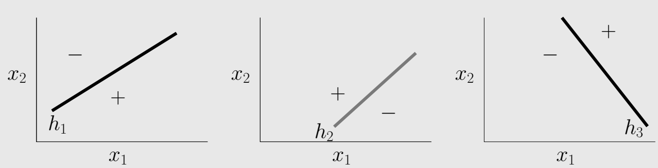
$$f = h_1\overline{h_2} + \overline{h_1}h_2.$$

Exercise 7.1

Consider a target function f whose '+' and '−' regions are illustrated below.



The target f has three perceptron components h_1, h_2, h_3:



Show that

$$f = \overline{h_1}h_2h_3 + h_1\overline{h_2}h_3 + h_1h_2\overline{h_3}.$$

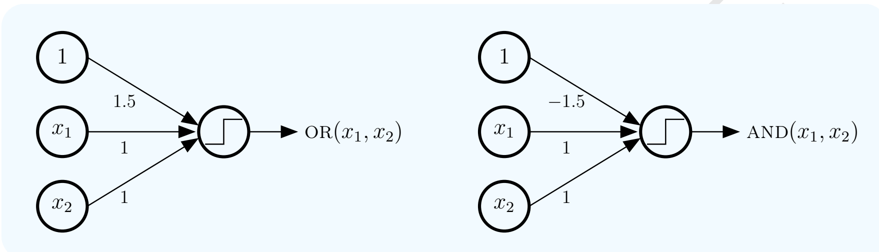
Is there a systematic way of going from a target which is a decomposition of perceptrons to a Boolean formula like this? *[Hint: consider only the regions of f which are '+' and use the* disjunctive normal form *(OR of ANDs).]*

Exercise 7.1 shows that a complicated target, which is composed of perceptrons, is a disjunction of conjunctions (OR of ANDs) applied to the component

perceptrons. This is a useful insight, because OR and AND can be implemented by the perceptron:

$$\begin{aligned}\text{OR}(x_1, x_2) &= \text{sign}(x_1 + x_2 + 1.5);\\ \text{AND}(x_1, x_2) &= \text{sign}(x_1 + x_2 - 1.5).\end{aligned}$$

This implies that these more complicated targets are ultimately just combinations of perceptrons. To see how to combine the perceptrons to get f, we introduce a graph representation of perceptrons, starting with OR and AND:



A node outputs a value to an arrow. The weight on an arrow multiplies this output and passes the result to the next node. Everything coming to this next node is summed and then transformed by $\text{sign}(\cdot)$ to get the final output.

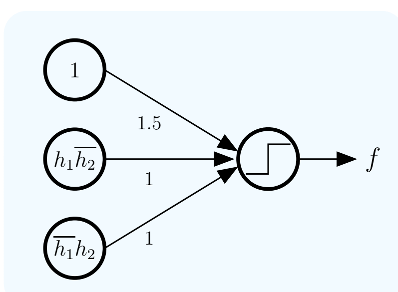
Exercise 7.2

(a) The Boolean OR and AND of two inputs can be extended to more than two inputs: $\text{OR}(x_1, \ldots, x_M) = +1$ if any one of the M inputs is $+1$; $\text{AND}(x_1, \ldots, x_M) = +1$ if all the inputs equal $+1$. Give graph representations of $\text{OR}(x_1, \ldots, x_M)$ and $\text{AND}(x_1, \ldots, x_M)$.

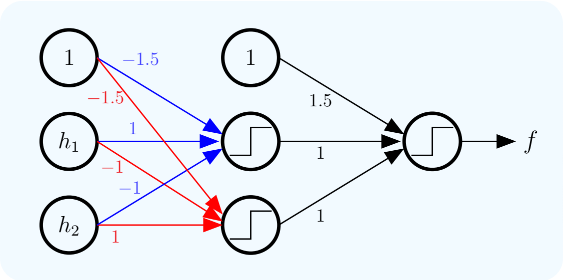
(b) Give the graph representation of the perceptron: $h(\mathbf{x}) = \text{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x})$.

(c) Give the graph representation of $\text{OR}(x_1, \overline{x_2}, x_3)$.

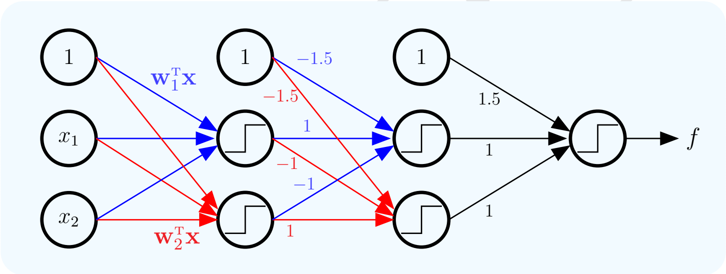
The MLP for a Complex Target. Since $f = h_1\overline{h_2} + \overline{h_1}h_2$, which is an OR of the two inputs $h_1\overline{h_2}$ and $\overline{h_1}h_2$, we first use the OR perceptron, to obtain:



The two inputs $h_1\overline{h_2}$ and $\overline{h_1}h_2$ are ANDs. As such, they can be simulated by the output of two AND perceptrons. To deal with negation of the inputs to the AND, we negate the weights multiplying the negated inputs (as you have done in Exercise 7.1(c)). The resulting graph representation of f is:



The blue and red weights are simulating the required two ANDs. Finally, since $h_1 = \text{sign}(\mathbf{w}_1^{\mathrm{T}}\mathbf{x})$ and $h_2 = \text{sign}(\mathbf{w}_2^{\mathrm{T}}\mathbf{x})$ are perceptrons, we further expand the h_1 and h_2 nodes to obtain the graph representation of f.



The next exercise asks you to compute an explicit algebraic formula for f. The visual graph representation is much neater and easier to generalize.

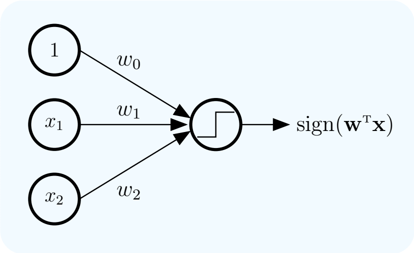
Exercise 7.3

Use the graph representation to get an explicit formula for f and show that:

$$f(\mathbf{x}) = \text{sign}\Big[\text{sign}(h_1(\mathbf{x}) - h_2(\mathbf{x}) - \tfrac{3}{2}) - \text{sign}(h_1(\mathbf{x}) - h_2(\mathbf{x}) + \tfrac{3}{2}) + \tfrac{3}{2}\Big],$$

where $h_1(\mathbf{x}) = \text{sign}(\mathbf{w}_1^{\mathrm{T}}\mathbf{x})$ and $h_2(\mathbf{x}) = \text{sign}(\mathbf{w}_2^{\mathrm{T}}\mathbf{x})$

Let's compare the graph form of f with the graph form of the simple perceptron, shown to the right. More layers of nodes are used between the input and output to implement f, as compared to the simple perceptron, hence we call it a multi-layer layer perceptron (MLP),. The additional layers are called *hidden layers*.

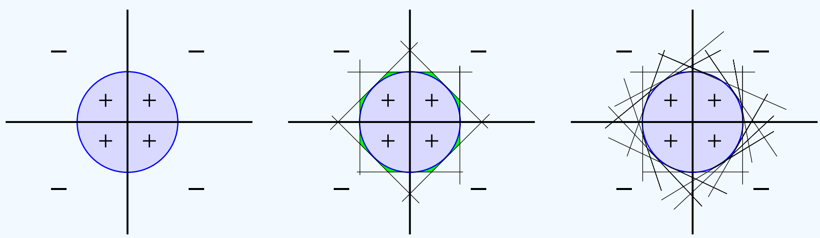


Notice that the layers feed forward into the next layer only (there are no backward pointing arrows and no jumps to other layers). The input (leftmost) layer is not counted as a layer, so in this example, there are 3 layers (2 hidden layers with 3 nodes each, and an output layer with 1 node). The simple perceptron has no hidden layers, just an input and output. The addition of hidden layers is what allowed us to implement the more complicated target.

Exercise 7.4

For the target function in Exercise 7.1, give the MLP in graphical form, as well as the explicit algebraic form.

If f can be decomposed into perceptrons using an OR of ANDs, then it can be implemented by a 3-layer perceptron. If f is not strictly decomposable into perceptrons, but the decision boundary is smooth, then a 3-layer perceptron can come arbitrarily close to implementing f. A 'proof by picture' illustration for a disc target function follows:



Target 8 perceptrons 16 perceptrons

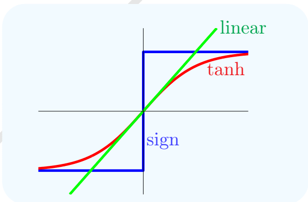
The formal proof is somewhat analogous to the theorem in calculus which says that any continuous function on a compact set can be approximated arbitrarily closely using step functions. The perceptron is the analog of the step function.

We have thus found a generalization of the simple perceptron that looks much like the simple perceptron itself, except for the addition of more layers. We gained the ability to model more complex target functions by adding more nodes (*hidden units*) in the hidden layers – this corresponds to allowing more perceptrons in the decomposition of f. In fact, a suitably large 3-layer MLP can closely approximate just about any target function, and fit any data set, so it is a very powerful learning model. Use it with care. If your MLP is too large you may lose generalization ability.

Once you fix the size of the MLP (number of hidden layers and number of hidden units in each layer), you learn the weights on every link (arrow) by fitting the data. Let's consider the simple perceptron,

$$h(\mathbf{x}) = \theta(\mathbf{w}^{\mathrm{T}}\mathbf{x}).$$

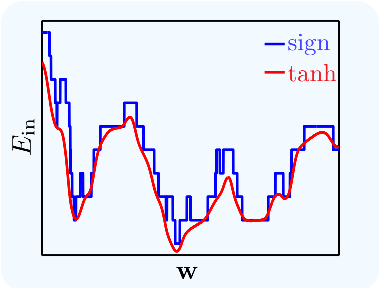
When $\theta(s) = \text{sign}(s)$, learning the weights was already a hard combinatorial problem and had a variety of algorithms, including the pocket algorithm, for fitting data (Chapter 3). The combinatorial optimization problem is even harder with the MLP, for the same reason, namely that the $\text{sign}(\cdot)$ function is not smooth; a smooth, differentiable approximation to $\text{sign}(\cdot)$ will allow us to use analytic methods, rather than purely combinatorial methods, to find the optimal weights. We therefore approximate, or 'soften' the $\text{sign}(\cdot)$ function by using the $\tanh(\cdot)$ function. The MLP is sometimes called a (hard) threshold neural network because the transformation function is a hard threshold at zero.



Here, we choose $\theta(x) = \tanh(x)$ which is in-between linear and the hard threshold: nearly linear for $x \approx 0$ and nearly ± 1 for $|x|$ large. The $\tanh(\cdot)$ function is another example of a *sigmoid* (because its shape looks like a flattened out 's'), related to the sigmoid we used for logistic regression.[2] Such networks are called sigmoidal neural networks. Just as we could use the weights learned from linear regression for classification, we could use weights learned using the sigmoidal neural network with $\tanh(\cdot)$ activation function for classification by replacing the output activation function with $\text{sign}(\cdot)$.

Exercise 7.5

Given $\mathbf{w}_1$ and $\epsilon > 0$, find $\mathbf{w}_2$ such that $|\text{sign}(\mathbf{w}_1^{\mathrm{T}}\mathbf{x}_n) - \tanh(\mathbf{w}_2^{\mathrm{T}}\mathbf{x}_n)| \le \epsilon$ for $\mathbf{x}_n \in \mathcal{D}$. *[Hint: For large enough α,* $\text{sign}(x) \approx \tanh(\alpha x)$.*]*

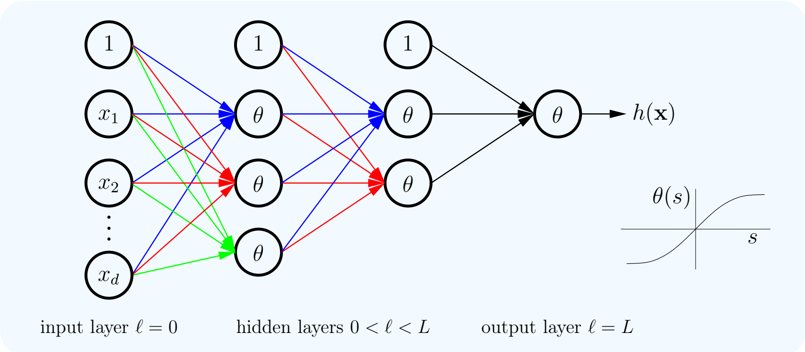


The previous example shows that the $\text{sign}(\cdot)$ function can be closely approximated by the $\tanh(\cdot)$ function. A concrete illustration of this is shown in the figure to the right. The figure shows how the in-sample error E_{in} varies with one of the weights in $\mathbf{w}$ on an example problem for the perceptron (blue curve) as compared to the sigmoidal version (red curve). The sigmoidal approximation captures the general shape of the error, so that if we minimize the sigmoidal in-sample error, we get a good approximation to minimizing the in-sample classification error.

[2] In logistic regression, we used the sigmoid because we wanted a probability as the output. Here, we use the 'soft' $\tanh(\cdot)$ because we want a friendly objective function to optimize.

7.2 Neural Networks

The neural network is our 'softened' MLP. Let's begin with a graph representation of a *feed-forward neural network* (the only kind we will consider).



The graph representation depicts a function in our hypothesis set. While this graphical view is aesthetic and intuitive, with information 'flowing' from the inputs on the far left, along links and through hidden nodes, ultimately to the output $h(\mathbf{x})$ on the far right, it will be necessary to algorithmically describe the function being computed. Things are going to get messy, and this calls for a very systematic notation; bear with us.

7.2.1 Notation

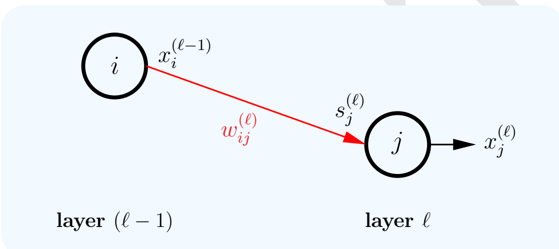
There are layers labeled by $\ell = 0, 1, 2, \ldots, L$. In our example above, $L = 3$, i.e. we have three layers (the input layer $\ell = 0$ is usually not considered a layer and is meant for feeding in the inputs). The layer $\ell = L$ is the output layer, which determines the value of the function. The layers in between, $0 < \ell < L$, are the hidden layers. We will use superscript$^{(\ell)}$ to refer to a particular layer. Each layer ℓ has 'dimension' $d^{(\ell)}$, which means that it has $d^{(\ell)} + 1$ nodes, labeled $0, 1, \ldots, d^{(\ell)}$. Every layer has one special node, which is called the *bias* node (labeled 0). This bias node is set to have an output 1, which is analogous to the fictitious $x_0 = 1$ convention that we had for linear models.

Every arrow represents a weight or connection strength from a node in a layer to a node in the *next higher* layer. Notice that the bias nodes have no incoming weights. There are no other connection weights.[3] A node with an incoming weight indicates that some signal is fed into this node. Every such node with an input has a *transformation function* θ. If $\theta(s) = \text{sign}(s)$, then we have the MLP for classification. As we mentioned before, we will be using a soft version of the MLP with $\theta(x) = \tanh(x)$ to approximate the $\text{sign}(\cdot)$ function. The $\tanh(\cdot)$ is a soft threshold or sigmoid, and we already saw a related sigmoid

[3] In a more general setting, weights can connect any two nodes, in addition to going backward (i.e., one can have cycles). Such networks are called recurrent neural networks, and we do not consider them here.

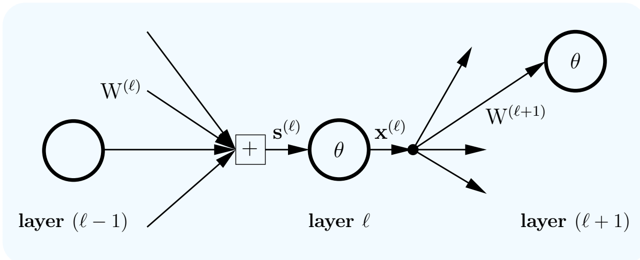
when we discussed logistic regression in Chapter 3. Ultimately, when we do classification, we replace the output sigmoid by the hard threshold sign$(\cdot)$. As a comment, if we were doing regression instead, our entire discussion goes through with the output transformation being replaced by the identity function (no transformation) so that the output is a real number. If we were doing logistic regression, we would replace the output $\tanh(\cdot)$ sigmoid by the logistic regression sigmoid.

The neural network model $\mathcal{H}_{\text{nn}}$ is specified once you determine the *architecture* of the neural network, that is the dimension of each layer $\mathbf{d} = [d^{(0)}, d^{(1)}, \ldots, d^{(L)}]$ (L is the number of layers). A hypothesis $h \in \mathcal{H}_{\text{nn}}$ is specified by selecting weights for the links. Let's zoom into a node in hidden layer ℓ, to see what weights need to be specified.



A node has an incoming signal s and an output x. The weights on links into the node from the previous layer are $w^{(\ell)}$, so the weights are indexed by the layer into which they go. Thus, the output of the nodes in layer $\ell - 1$ is multiplied by weights $w^{(\ell)}$. We use subscripts to index the nodes in a layer. So, $w_{ij}^{(\ell)}$ is the weight *into* node j in layer ℓ *from* node i in the previous layer, the signal going into node j in layer ℓ is $s_j^{(\ell)}$, and the output of this node is $x_j^{(\ell)}$. There are some special nodes in the network. The zero nodes in every layer are constant nodes, set to output 1. They have no incoming weight, but they have an outgoing weight. The nodes in the input layer $\ell = 0$ are for the input values, and have no incoming weight or transformation function.

For the most part, we only need to deal with the network on a layer by layer basis, so we introduce vector and matrix notation for that. We collect all the input signals to nodes $1, \ldots, d^{(\ell)}$ in layer ℓ in the vector $\mathbf{s}^{(\ell)}$. Similarly, collect the output from nodes $0, \ldots, d^{(\ell)}$ in the vector $\mathbf{x}^{(\ell)}$; note that $\mathbf{x}^{(\ell)} \in \{1\} \times \mathbb{R}^{d^{(\ell)}}$ because of the bias node 0. There are links connecting the outputs of all nodes in the previous layer to the inputs of layer ℓ. So, into layer ℓ, we have a $(d^{(\ell-1)} + 1) \times d^{(\ell)}$ matrix of weights $\mathrm{W}^{(\ell)}$. The (i, j)-entry of $\mathrm{W}^{(\ell)}$ is $w_{ij}^{(\ell)}$ going from node i in the previous layer to node j in layer ℓ.



layer ℓ parameters

signals in	$\mathbf{s}^{(\ell)}$	$d^{(\ell)}$ dimensional input vector
outputs	$\mathbf{x}^{(\ell)}$	$d^{(\ell)}+1$ dimensional output vector
weights in	$\mathrm{W}^{(\ell)}$	$(d^{(\ell-1)}+1)\times d^{(\ell)}$ dimensional matrix
weights out	$\mathrm{W}^{(\ell+1)}$	$(d^{(\ell)}+1)\times d^{(\ell+1)}$ dimensional matrix

After you fix the weights $\mathrm{W}^{(\ell)}$ for $\ell = 1, \ldots, L$, you have specified a particular neural network hypothesis $h \in \mathcal{H}_{\text{nn}}$. We collect all these weight matrices into a single weight parameter $\mathbf{w} = \{\mathrm{W}^{(1)}, \mathrm{W}^{(2)}, \ldots, \mathrm{W}^{(L)}\}$, and sometimes we will write $h(\mathbf{x}; \mathbf{w})$ to explicitly indicate the dependence of the hypothesis on $\mathbf{w}$.

7.2.2 Forward Propagation

The neural network hypothesis $h(\mathbf{x})$ is computed by the *forward propagation* algorithm. First observe that the inputs and outputs of a layer are related by the transformation function,

$$\mathbf{x}^{(\ell)} = \begin{bmatrix} 1 \\ \theta(\mathbf{s}^{(\ell)}) \end{bmatrix}. \tag{7.1}$$

where $\theta(\mathbf{s}^{(\ell)})$ is a vector whose components are $\theta(s_j^{(\ell)})$. To get the input vector into layer ℓ, we compute the weighted sum of the outputs from the previous layer, with weights specified in $\mathrm{W}^{(\ell)}$: $s_j^{(\ell)} = \sum_{i=0}^{d^{(\ell-1)}} w_{ij}^{(\ell)} x_i^{(\ell-1)}$. This process is compactly represented by the matrix equation

$$\mathbf{s}^{(\ell)} = (\mathrm{W}^{(\ell)})^{\mathrm{T}} \mathbf{x}^{(\ell-1)}. \tag{7.2}$$

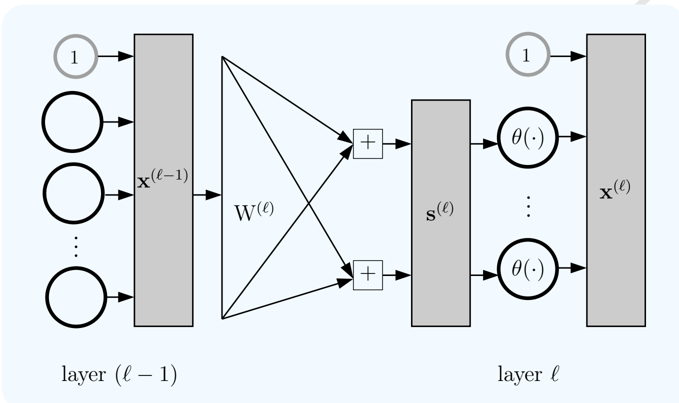
All that remains is to initialize the input layer to $\mathbf{x}^{(0)} = \mathbf{x}$ (so $d^{(0)} = d$, the input dimension)[4] and use Equations (7.2) and (7.1) in the following chain,

$$\mathbf{x} = \mathbf{x}^{(0)} \xrightarrow{\mathrm{W}^{(1)}} \mathbf{s}^{(1)} \xrightarrow{\theta} \mathbf{x}^{(1)} \xrightarrow{\mathrm{W}^{(2)}} \mathbf{s}^{(2)} \xrightarrow{\theta} \mathbf{x}^{(2)} \cdots \longrightarrow \mathbf{s}^{(L)} \xrightarrow{\theta} \mathbf{x}^{(L)} = h(\mathbf{x}).$$

[4] Recall that the input vectors are also augmented with $x_0 = 1$.

Forward propagation to compute $h(\mathbf{x})$:

$\mathbf{x}^{(0)} \leftarrow \mathbf{x}$ **[Initialization]**
for $\ell = 1$ to L **do** **[Forward Propagation]**
  $\mathbf{s}^{(\ell)} \leftarrow (\mathrm{W}^{(\ell)})^{\mathrm{T}}\mathbf{x}^{(\ell-1)}$
  $\mathbf{x}^{(\ell)} \leftarrow \begin{bmatrix} 1 \\ \theta(\mathbf{s}^{(\ell)}) \end{bmatrix}$
$h(\mathbf{x}) = \mathbf{x}^{(L)}$ **[Output]**



After forward propagation, the output vector $\mathbf{x}^{(\ell)}$ at every layer $l = 0, \ldots, L$ has been computed.

Exercise 7.6

Let V and Q be the number of nodes and weights in the neural network,

$$V = \sum_{\ell=0}^{L} d^{(\ell)}, \qquad Q = \sum_{\ell=1}^{L} d^{(\ell)}(d^{(\ell-1)} + 1).$$

In terms of V and Q, how many computations are made in forward propagation (additions, multiplications and evaluations of θ).

[Answer: $O(Q)$ multiplications and additions, and $O(V)$ θ-evaluations.]

If we want to compute E_{in}, all we need is $h(\mathbf{x}_n)$ and y_n. For the sum of squares,

$$\begin{aligned} E_{\text{in}}(\mathbf{w}) &= \frac{1}{N}\sum_{n=1}^{N}(h(\mathbf{x}_n; \mathbf{w}) - y_n)^2 \\ &= \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n^{(L)} - y_n)^2. \end{aligned}$$

We now discuss how to minimize E_{in} to obtain the learned weights. It will be a direct application of gradient descent, with a special algorithm that computes the gradient efficiently.

7.2.3 Backpropagation Algorithm

We studied an algorithm for getting to a local minimum of a smooth in-sample error surface in Chapter 3, namely gradient descent: initialize the weights to $\mathbf{w}(0)$ and for $t = 1, 2, \ldots$ update the weights by taking a step in the negative gradient direction,

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla E_{\text{in}}(\mathbf{w}(t))$$

we called this (batch) *gradient descent*. To implement gradient descent, we need the gradient.

Exercise 7.7

For the sigmoidal perceptron, $h(\mathbf{x}) = \tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x})$, let the in-sample error be $E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}(\tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) - y_n)^2$. Show that

$$\nabla E_{\text{in}}(\mathbf{w}) = \frac{2}{N}\sum_{n=1}^{N}(\tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) - y_n)(1 - \tanh^2(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n))\mathbf{x}_n.$$

If $\mathbf{w} \to \infty$, what happens to the gradient; how this is related to why it is hard to optimize the perceptron.

We now consider the sigmoidal multi-layer neural network with $\theta(x) = \tanh(x)$. Since $h(\mathbf{x})$ is smooth, we can apply gradient descent to the resulting error function. To do so, we need the gradient $\nabla E_{\text{in}}(\mathbf{w})$. Recall that the weight vector $\mathbf{w}$ contains all the weight matrices $\mathrm{W}^{(1)}, \ldots, \mathrm{W}^{(L)}$, and we need the derivatives with respect to all these weights. Unlike the sigmoidal perceptron in Exercise 7.7, for the multilayer sigmoidal network there is no simple closed form expression for the gradient. Consider an in-sample error which is the sum of the point-wise errors over the data points (as is the squared in-sample error),

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathsf{e}_n.$$

where $\mathsf{e}_n = \mathsf{e}(h(\mathbf{x}_n), y_n)$. For the squared error, $\mathsf{e}(h, y) = (h - y)^2$. To compute the gradient of E_{in}, we need its derivative with respective to each weight matrix:

$$\frac{\partial E_{\text{in}}}{\partial \mathrm{W}^{(\ell)}} = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial \mathsf{e}_n}{\partial \mathrm{W}^{(\ell)}}, \tag{7.3}$$

The basic building block in (7.3) is the partial derivative of the error on a data point e, with respect to the $\mathrm{W}^{(\ell)}$. A quick and dirty way to get $\frac{\partial \mathsf{e}}{\partial \mathrm{W}^{(\ell)}}$ is

to use the numerical finite difference approach. The complexity of obtaining the partial derivatives with respect to every weight is $O(Q^2)$, where Q is the number of weights (see Problem 7.6). From (7.3), we have to compute these derivatives for every data point, so the numerical approach is computationally prohibitive. We now derive an elegant dynamic programming algorithm known as *backpropagation*.[5] Backpropagation allows us to compute the partial derivatives with respect to every weight efficiently, using $O(Q)$ computation. We describe backpropagation for getting the partial derivative of the error e, but the algorithm is general and can be used to get the partial derivative of any function of the output $h(\mathbf{x})$ with respect to the weights.

Backpropagation is based on several applications of the chain rule to write partial derivatives in layer ℓ using partial derivatives in layer $(\ell+1)$. To describe the algorithm, we define the *sensitivity vector* for layer ℓ, which is the sensitivity (gradient) of the error e with respect to the input signal $\mathbf{s}^{(\ell)}$ that goes into layer ℓ. We denote the sensitivity by $\boldsymbol{\delta}^{(\ell)}$,

$$\boldsymbol{\delta}^{(\ell)} = \frac{\partial \mathsf{e}}{\partial \mathbf{s}^{(\ell)}}.$$

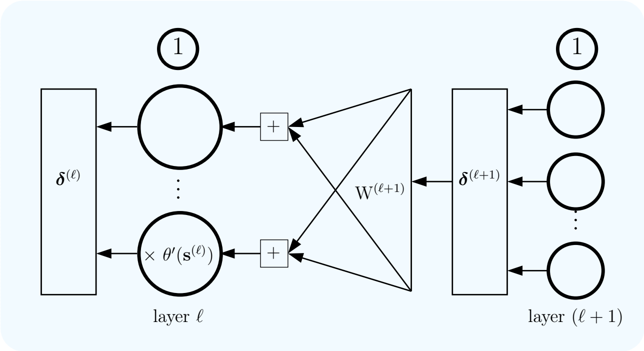
The sensitivity quantifies how e changes with $\mathbf{s}^{(\ell)}$. Using the sensitivity, we can write the partial derivatives with respect to the weights $\mathrm{W}^{(\ell)}$ as

$$\frac{\partial \mathsf{e}}{\partial \mathrm{W}^{(\ell)}} = \mathbf{x}^{(\ell-1)}(\boldsymbol{\delta}^{(\ell)})^{\mathrm{T}}. \tag{7.4}$$

We will derive this formula later, but for now let's examine it closely. The partial derivatives on the left form a matrix with dimensions $(d^{(\ell-1)}+1)\times d^{(\ell)}$ and the 'outer product' of the two vectors on the right give exactly such a matrix. The partial derivatives have contributions from two components. (i) The output vector of the layer from which the weights originate; the larger the output, the more sensitive e is to the weights in the layer. (ii) The sensitivity vector of the layer into which the weights go; the larger the sensitivity vector, the more sensitive e is to the weights in that layer.

The outputs $\mathbf{x}^{(\ell)}$ for every layer $\ell \geq 0$ can be computed by a forward propagation. So to get the partial derivatives, it suffices to obtain the sensitivity vectors $\boldsymbol{\delta}^{(\ell)}$ for every layer $\ell \geq 1$ (remember that there is no input signal to layer $\ell = 0$). It turns out that the sensitivity vectors can be obtained by running a slightly modified version of the neural network *backwards*, and hence the name backpropagation. In forward propagation, each layer outputs the vector $\mathbf{x}^{(\ell)}$ and in backpropagation, each layer outputs (backwards) the vector $\boldsymbol{\delta}^{(\ell)}$. In forward propagation, we compute $\mathbf{x}^{(\ell)}$ from $\mathbf{x}^{(\ell-1)}$ and in backpropagation, we compute $\boldsymbol{\delta}^{(\ell)}$ from $\boldsymbol{\delta}^{(\ell+1)}$. The basic idea is illustrated in the following figure.

[5] Dynamic programming is an elegant algorithmic technique in which one builds up a solution to a complex problem using the solutions to related but simpler problems.



As you can see in the figure, the neural network is slightly modified only in that we have changed the transformation function for the nodes. In forward propagation, the transformation was the sigmoid $\theta(\cdot)$. In backpropagation, the transformation is *multiplication by* $\theta'(\mathbf{s}^{(\ell)})$, where $\mathbf{s}^{(\ell)}$ is the input to the node. So the transformation function is now different for each node, and it depends on the input to the node, which depends on $\mathbf{x}$. This input was computed in the forward propagation. For the $\tanh(\cdot)$ transformation function, $\tanh'(\mathbf{s}^{(\ell)}) = \mathbf{1} - \tanh^2(\mathbf{s}^{(\ell)}) = \mathbf{1} - \mathbf{x}^{(\ell)} \otimes \mathbf{x}^{(\ell)}$, where $\otimes$ denotes component-wise multiplication.

In the figure, layer $(\ell+1)$ outputs (backwards) the sensitivity vector $\boldsymbol{\delta}^{(\ell+1)}$, which gets multiplied by the weights $W^{(\ell+1)}$, summed and passed into the nodes in layer ℓ. Nodes in layer ℓ multiply by $\theta'(\mathbf{s}^{(\ell)})$ to get $\boldsymbol{\delta}^{(\ell)}$. Using $\otimes$, a shorthand notation for this backpropagation step is:

$$\boldsymbol{\delta}^{(\ell)} = \theta'(\mathbf{s}^{(\ell)}) \otimes \left[W^{(\ell+1)}\boldsymbol{\delta}^{(\ell+1)}\right]_1^{d^{(\ell)}}, \tag{7.5}$$

where the vector $\left[W^{(\ell+1)}\boldsymbol{\delta}^{(\ell+1)}\right]_1^{d^{(\ell)}}$ contains components $1, \ldots, d^{(\ell)}$ of the vector $W^{(\ell+1)}\boldsymbol{\delta}^{(\ell+1)}$ (excluding the bias component which has index 0). This formula is not surprising. The sensitivity of e to inputs of layer ℓ is proportional to the slope of the activation function in layer ℓ (bigger slope means a small change in $\mathbf{s}^{(\ell)}$ will have a larger effect on $\mathbf{x}^{(\ell)}$), the size of the weights going out of the layer (bigger weights mean a small change in $\mathbf{s}^{(\ell)}$ will have more impact on $\mathbf{s}^{(\ell+1)}$) and the sensitivity in the next layer (a change in layer ℓ affects the inputs to layer $\ell+1$, so if e is more sensitive to layer $\ell+1$, then it will also be more sensitive to layer ℓ).

We will derive this backward recursion later. For now, observe that if we know $\boldsymbol{\delta}^{(\ell+1)}$, then you can get $\boldsymbol{\delta}^{(\ell)}$. We use $\boldsymbol{\delta}^{(L)}$ to seed the backward process, and we can get that explicitly because $\mathsf{e} = (\mathbf{x}^{(L)} - y)^2 = (\theta(\mathbf{s}^{(L)}) - y)^2$.

Therefore,

$$\begin{aligned}\boldsymbol{\delta}^{(L)} &= \frac{\partial \mathsf{e}}{\partial \mathbf{s}^{(L)}}\\ &= \frac{\partial}{\partial \mathbf{s}^{(L)}}(\mathbf{x}^{(L)}-y)^2\\ &= 2(\mathbf{x}^{(L)}-y)\frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}}\\ &= 2(\mathbf{x}^{(L)}-y)\theta'(\mathbf{s}^{(L)}).\end{aligned}$$

When the output transformation is $\tanh(\cdot)$, $\theta'(\mathbf{s}^{(L)}) = 1-(x^{(L)})^2$ (classification); when the output transformation is the identity (regression), $\theta'(\mathbf{s}^{(L)}) = 1$. Now, using (7.5), we can compute all the sensitivities:

$$\boldsymbol{\delta}^{(1)} \longleftarrow \boldsymbol{\delta}^{(2)} \cdots \longleftarrow \boldsymbol{\delta}^{(L-1)} \longleftarrow \boldsymbol{\delta}^{(L)}.$$

Note that since there is only one output node, $\mathbf{s}^L$ is a scalar, and so too is $\boldsymbol{\delta}^{(L)}$. The algorithm box below summarizes backpropagation.

Backpropagation to compute sensitivities $\boldsymbol{\delta}^{(\ell)}$.
Input: a data point $(\mathbf{x}, y)$.

0: Run forward propagation on $\mathbf{x}$ to compute and save:

$$\begin{array}{ll}\mathbf{s}^{(\ell)} & \text{for } \ell = 1,\ldots,L;\\ \mathbf{x}^{(\ell)} & \text{for } \ell = 0,\ldots,L.\end{array}$$

1: $\delta^{(L)} \leftarrow 2(x^{(L)}-y)\theta'(s^{(L)})$ **[Initialization]**

$$\theta'(s^{(L)}) = \begin{cases}1-(x^{(L)})^2 & \theta(s) = \tanh(s);\\ 1 & \theta(s) = s.\end{cases}$$

2: **for** $\ell = L-1$ to 1 **do** **[Back-Propagation]**

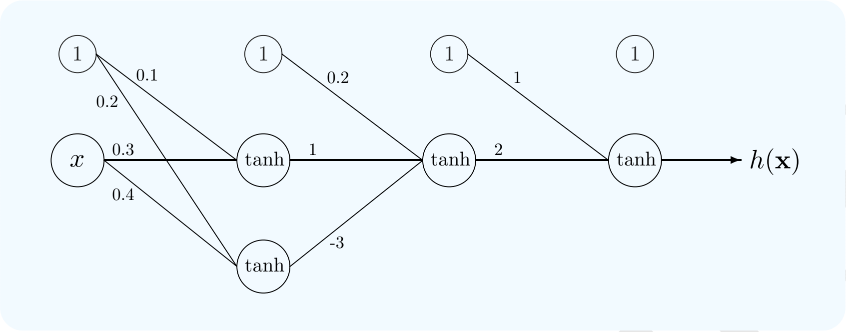
3: Let $\theta'(\mathbf{s}^{(\ell)}) = \left[1-\mathbf{x}^{(\ell)}\otimes\mathbf{x}^{(\ell)}\right]_1^{d^{(\ell)}}$.

4: Compute the sensitivity $\boldsymbol{\delta}^{(\ell)}$ from $\boldsymbol{\delta}^{(\ell+1)}$:

$$\boldsymbol{\delta}^{(\ell)} \leftarrow \theta'(\mathbf{s}^{(\ell)}) \otimes \left[\mathrm{W}^{(\ell+1)}\boldsymbol{\delta}^{(\ell+1)}\right]_1^{d^{(\ell)}}$$

In step 3, we assumed tanh-hidden node transformations. If the hidden unit transformation functions are not $\tanh(\cdot)$, then the derivative in step 3 should be updated accordingly. Using forward propagation, we compute $\mathbf{x}^{(\ell)}$ for $\ell = 0,\ldots,L$ and using backpropagation, we compute $\boldsymbol{\delta}^{(\ell)}$ for $\ell = 1,\ldots,L$. Finally, we get the partial derivative of the error on a *single* data point using Equation (7.4). Nothing illuminates the moving parts better than working an example from start to finish.

Example 7.1. Consider the following neural network.



There is a single input, and the weight matrices are:

$$\mathrm{W}^{(1)} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix}; \quad \mathrm{W}^{(2)} = \begin{bmatrix} 0.2 \\ 1 \\ -3 \end{bmatrix}; \quad \mathrm{W}^{(3)} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

For the data point $x = 2, y = 1$, forward propagation gives:

$\mathbf{x}^{(0)}$	$\mathbf{s}^{(1)}$	$\mathbf{x}^{(1)}$	$\mathbf{s}^{(2)}$	$\mathbf{x}^{(2)}$	$\mathbf{s}^{(3)}$	$\mathbf{x}^{(3)}$
$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$	$\begin{bmatrix} 0.1 & 0.3 \\ 0.2 & 0.4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0.7 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0.60 \\ 0.76 \end{bmatrix}$	$[-1.48]$	$\begin{bmatrix} 1 \\ -0.90 \end{bmatrix}$	$[-0.8]$	-0.66

We show above how $\mathbf{s}^{(1)} = (\mathrm{W}^{(1)})^{\mathrm{T}} \mathbf{x}^{(0)}$ is computed. Backpropagation gives

$\boldsymbol{\delta}^{(3)}$	$\boldsymbol{\delta}^{(2)}$	$\boldsymbol{\delta}^{(1)}$
$[-1.855]$	$[(1 - 0.9^2) \cdot 2 \cdot -1.855] = [-0.69]$	$\begin{bmatrix} -0.44 \\ 0.88 \end{bmatrix}$

We have explicitly shown how $\boldsymbol{\delta}^{(2)}$ is obtained from $\boldsymbol{\delta}^{(3)}$. It is now a simple matter to combine the output vectors $\mathbf{x}^{(\ell)}$ with the sensitivity vectors $\boldsymbol{\delta}^{(\ell)}$ using (7.4) to obtain the partial derivatives that are needed for the gradient:

$$\frac{\partial \mathrm{e}}{\partial \mathrm{W}^{(1)}} = \mathbf{x}^{(0)} (\boldsymbol{\delta}^{(1)})^{\mathrm{T}} = \begin{bmatrix} -0.44 & 0.88 \\ -0.88 & 1.75 \end{bmatrix}; \quad \frac{\partial \mathrm{e}}{\partial \mathrm{W}^{(2)}} = \begin{bmatrix} -0.69 \\ -0.42 \\ -0.53 \end{bmatrix}; \quad \frac{\partial \mathrm{e}}{\partial \mathrm{W}^{(3)}} = \begin{bmatrix} -1.85 \\ 1.67 \end{bmatrix}.$$

□

Exercise 7.8

Repeat the computations in Example 7.1 for the case when the output transformation is the identity. You should compute $\mathbf{s}^{(\ell)}, \mathbf{x}^{(\ell)}, \boldsymbol{\delta}^{(\ell)}$ and $\partial \mathrm{e}/\partial \mathrm{W}^{(\ell)}$

Let's derive (7.4) and (7.5), which are the core equations of backpropagation. There's nothing to it but repeated application of the chain rule. If you wish to trust our math, you won't miss much by moving on.

Begin safe skip: If you trust our math, you can skip this part without compromising the logical sequence. A similar green box will tell you when to rejoin.

To begin, let's take a closer look at the partial derivative, $\partial \mathrm{e}/\partial \mathrm{W}^{(\ell)}$. The situation is illustrated in Figure 7.1.

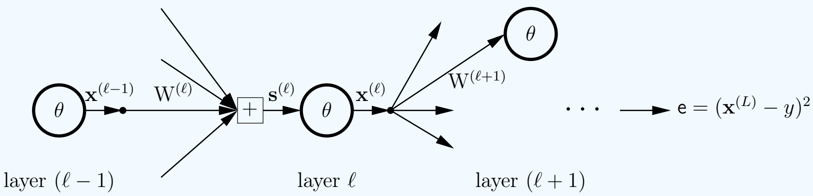


Figure 7.1: Chain of dependencies from $\mathrm{W}^{(\ell)}$ to $\mathbf{x}^{(L)}$.

We can identify the following chain of dependencies by which $\mathrm{W}^{(\ell)}$ influences the output $\mathbf{x}^{(L)}$, and hence the error e.

$$\mathrm{W}^{(\ell)} \longrightarrow \mathbf{s}^{(\ell)} \longrightarrow \mathbf{x}^{(\ell)} \longrightarrow \mathbf{s}^{(\ell+1)} \cdots \longrightarrow \mathbf{x}^{(L)} = h.$$

To derive (7.4), we drill down to a single weight and use the chain rule. For a single weight $w_{ij}^{(\ell)}$, a change in $w_{ij}^{(\ell)}$ only affects $\mathbf{s}_j^{(\ell)}$ and so by the chain rule,

$$\frac{\partial \mathrm{e}}{\partial w_{ij}^{(\ell)}} = \frac{\partial \mathbf{s}_j^{(\ell)}}{\partial w_{ij}^{(\ell)}} \cdot \frac{\partial \mathrm{e}}{\partial \mathbf{s}_j^{(\ell)}} = \mathbf{x}_i^{(\ell-1)} \cdot \boldsymbol{\delta}_j^{(\ell)},$$

where the last equality follows because $\mathbf{s}_j^{(\ell)} = \sum_{\alpha=0}^{d^{(\ell-1)}} w_{\alpha j}^{(\ell)} \mathbf{x}_\alpha^{(\ell-1)}$ and by definition of $\boldsymbol{\delta}_j^{(\ell)}$. We have derived the component form of (7.4).

We now derive the component form of (7.5). Since e depends on $\mathbf{s}^{(\ell)}$ only through $\mathbf{x}^{(\ell)}$ (see Figure 7.1), by the chain rule, we have:

$$\boldsymbol{\delta}_j^{(\ell)} = \frac{\partial \mathrm{e}}{\partial \mathbf{s}_j^{(\ell)}} = \frac{\partial \mathrm{e}}{\partial \mathbf{x}_j^{(\ell)}} \cdot \frac{\partial \mathbf{x}_j^{(\ell)}}{\partial \mathbf{s}_j^{(\ell)}} = \theta'\left(\mathbf{s}_j^{(\ell)}\right) \cdot \frac{\partial \mathrm{e}}{\partial \mathbf{x}_j^{(\ell)}}.$$

To get the partial derivative $\partial \mathrm{e}/\partial \mathbf{x}^{(\ell)}$, we need to understand how e changes due to changes in $\mathbf{x}^{(\ell)}$. Again, from Figure 7.1, a change in $\mathbf{x}^{(\ell)}$ only affects

$\mathbf{s}^{(\ell+1)}$ and hence e. Because a particular component of $\mathbf{x}^{(\ell)}$ can affect *every* component of $\mathbf{s}^{(\ell+1)}$, we need to sum these dependencies using the chain rule:

$$\frac{\partial \mathsf{e}}{\partial \mathbf{x}_j^{(\ell)}} = \sum_{k=1}^{d^{(\ell+1)}} \frac{\partial \mathbf{s}_k^{(\ell+1)}}{\partial \mathbf{x}_j^{(\ell)}} \cdot \frac{\partial \mathsf{e}}{\partial \mathbf{s}_k^{(\ell+1)}} = \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(\ell+1)} \boldsymbol{\delta}_k^{(\ell+1)}.$$

Putting all this together, we have arrived at the component version of (7.5)

$$\boldsymbol{\delta}_j^{(\ell)} = \theta'(\mathbf{s}_j^{(\ell)}) \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(\ell+1)} \boldsymbol{\delta}_k^{(\ell+1)}, \tag{7.6}$$

Intuitively, the first term comes from the impact of $\mathbf{s}^{(\ell)}$ on $\mathbf{x}^{(\ell)}$; the summation is the impact of $\mathbf{x}^{(\ell)}$ on $\mathbf{s}^{(\ell+1)}$, and the impact of $\mathbf{s}^{(\ell+1)}$ on h is what gives us back the sensitivities in layer $(\ell+1)$, resulting in the backward recursion.

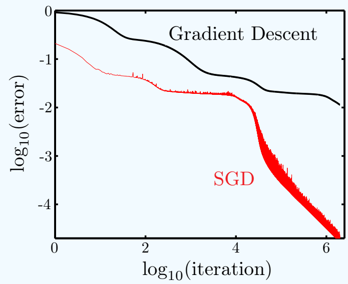
End safe skip: Those who skipped are now rejoining us to discuss how backpropagation gives us ∇E_{in}.

Backpropagation works with a data point $(\mathbf{x}, y)$ and weights $\mathbf{w} = \{\mathrm{W}^{(1)}, \ldots, \mathrm{W}^{(L)}\}$. Since we run one forward and backward propagation to compute the outputs $\mathbf{x}^{(\ell)}$ and the sensitivities $\boldsymbol{\delta}^{(\ell)}$, the running time is order of the number of weights in the network. We compute once for each data point $(\mathbf{x}_n, y_n)$ to get $\nabla E_{\text{in}}(\mathbf{x}_n)$ and, using the sum in (7.3), we aggregate these single point gradients to get the full *batch* gradient ∇E_{in}. We summarize the algorithm below.

Algorithm to Compute $E_{\text{in}}(\mathbf{w})$ and $\mathbf{g} = \nabla E_{\text{in}}(\mathbf{w})$.
Input: $\mathbf{w} = \{\mathrm{W}^{(1)}, \ldots, \mathrm{W}^{(L)}\}$; $\mathcal{D} = (\mathbf{x}_1, y_1) \ldots (\mathbf{x}_N, y_n)$.
Output: error $E_{\text{in}}(\mathbf{w})$ and gradient $\mathbf{g} = \{\mathrm{G}^{(1)}, \ldots, \mathrm{G}^{(L)}\}$.

Initialize: $E_{\text{in}} = 0$ and $\mathrm{G}^{(\ell)} = 0 \cdot \mathrm{W}^{(\ell)}$ for $\ell = 1, \ldots, L$.
for Each data point $(\mathbf{x}_n, y_n)$, $n = 1, \ldots, N$, **do**
  Compute $\mathbf{x}^{(\ell)}$ for $\ell = 0, \ldots, L$. [forward propagation]
  Compute $\boldsymbol{\delta}^{(\ell)}$ for $\ell = L, \ldots, 1$. [backpropagation]
  $E_{\text{in}} \leftarrow E_{\text{in}} + \frac{1}{N}(\mathbf{x}^{(L)} - y_n)^2$.
  for $\ell = 1, \ldots, L$ **do**
    $\mathrm{G}^{(\ell)}(\mathbf{x}_n) = [\mathbf{x}^{(\ell-1)}(\boldsymbol{\delta}^{(\ell)})^{\mathrm{T}}]$
    $\mathrm{G}^{(\ell)} \leftarrow \mathrm{G}^{(\ell)} + \frac{1}{N}\mathrm{G}^{(\ell)}(\mathbf{x}_n)$

($\mathrm{G}^{(\ell)}(\mathbf{x}_n)$ is the gradient on data point $\mathbf{x}_n$). The weight update for a single iteration of fixed learning rate gradient descent is $\mathrm{W}^{(\ell)} \leftarrow \mathrm{W}^{(\ell)} - \eta \mathrm{G}^{(\ell)}$, for $\ell = 1, \ldots, L$. We do all this for *one* iteration of gradient descent, a costly computation for just one little step.



In Chapter 3, we discussed *stochastic gradient descent (SGD)* as a more efficient alternative to the batch mode. Rather than wait for the aggregate gradient $\mathrm{G}^{(\ell)}$ at the end of the iteration, one immediately updates the weights as each data point is sequentially processed using the single point gradient in step 7 of the algorithm: $\mathrm{W}^{(\ell)} = \mathrm{W}^{(\ell)} - \eta \mathrm{G}^{(\ell)}(\mathbf{x}_n)$. In this sequential version, you still run a forward and backward propagation for each data point, but make N updates to the weights. A comparison of batch gradient descent with SGD is shown to the right. We used 500 training examples from the digits data and a 2-layer neural network with 5 hidden units and learning rate $\eta = 0.01$. The SGD curve is erratic because one is not minimizing the total error at each iteration, but the error on a specific data point. One method to dampen this erratic behavior is to decrease the learning rate as the minimization proceeds.

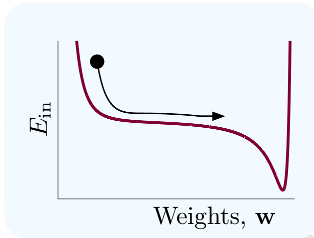
The speed at which you minimize E_{in} can depend heavily on the optimization algorithm you use. SGD appears significantly better than plain vanilla gradient descent, but we can do much better – even SGD is not very efficient. In Section 7.5, we discuss some other powerful methods (for example, conjugate gradients) that can significantly improve upon gradient descent and SGD, by making more effective use of the gradient.

Initialization and Termination. Choosing the initial weights and deciding when to stop the gradient descent can be tricky, as compared with logistic regression, because the in-sample error is not convex anymore. From Exercise 7.7, if the weights are initialized too large so that $\tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) \approx \pm 1$, then the gradient will be close to zero and the algorithm won't get anywhere. This is especially a problem if you happen to initialize the weights to the wrong sign. It is usually best to initialize the weights to *small* random values where $\tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) \approx 0$ so that the algorithm has the flexibility to move the weights easily to fit the data. One good choice is to initialize using Gaussian random weights, $w_i \sim N(0, \sigma_w^2)$ where σ_w^2 is small. But how small should σ_w^2 be? A simple heuristic is that we want $|\mathbf{w}^{\mathrm{T}}\mathbf{x}_n|^2$ to be small. Since $\mathbb{E}_{\mathbf{w}}\left[|\mathbf{w}^{\mathrm{T}}\mathbf{x}_n|^2\right] = \sigma_w^2 \|\mathbf{x}_n\|^2$, we should choose σ_w^2 so that $\sigma_w^2 \cdot \max_n \|\mathbf{x}_n\|^2 \ll 1$.

Exercise 7.9

What can go wrong if you just initialize all the weights to exactly zero?

When do we stop? It is risky to rely solely on the size of the gradient to stop. As illustrated on the right, you might stop prematurely when the iteration reaches a relatively flat region (which is more common than you might suspect). A combination of stopping criteria is best in practice, for example stopping only when there is marginal error improvement coupled with small error, plus an upper bound on the number of iterations.



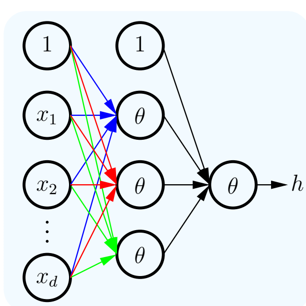
7.2.4 Regression for Classification

In Chapter 3, we mentioned that you could use the weights resulting from linear regression as perceptron weights for classification, and you can do the same with neural networks. Specifically, fit the classification data ($y_n = \pm 1$) as though it were a regression problem. This means you use the identity function as the output node transformation, instead of $\tanh(\cdot)$. This can be a great help because of the 'flat regions' which the network is susceptible to when using gradient descent, which happens often in training. The reason for these flat periods in the optimization is the exceptionally flat nature of the tanh function when its argument gets large. If for whatever reason the weights get large toward the beginning of the training, then the error surface begins to look flat, because the tanh has been saturated. Now, gradient descent cannot make any progress and you might think you are at a minimum, when in fact you are far from a minimum. The problem of a flat error surface is considerably mitigated when the output transformation is the identity because you can recover from an initial bad move if it happens to take you to large weights (the linear output never saturates). For a concrete example of a prematurely flat in-sample error, see the figure in Example 7.2 on page 25.

7.3 Approximation versus Generalization

A large enough MLP with 2 hidden layers can approximate smooth decision functions arbitrarily well. It turns out that a single hidden layer suffices.[6] A neural network with a single hidden layer having m hidden units ($d^{(1)} = m$) implements a function of the form

$$h(\mathbf{x}) = \theta\left(w_{01}^{(2)} + \sum_{j=1}^{m} w_{j1}^{(2)}\,\theta\left(\sum_{i=0}^{d} w_{ij}^{(1)} x_i\right)\right).$$



[6] Though one hidden layer is enough, it is not necessarily the most efficient way to fit the data; for example a much smaller 2-hidden-layer network may exist.

This is a cumbersome representation for such a simple network. A simplified notation for this special case is much more convenient. For the second-layer weights, we will just use $w_0, w_1, \ldots, w_m$ and we will use $\mathbf{v}_j$ to denote the jth column of the first layer weight matrix $\mathrm{W}^{(1)}$, for $j = 1 \ldots m$. With this simpler notation, the hypothesis becomes much more pleasant looking:

$$h(\mathbf{x}) = \theta\left(w_0 + \sum_{j=1}^{m} w_j \theta\left(\mathbf{v}_j^{\mathrm{T}}\mathbf{x}\right)\right).$$

Neural Network versus Nonlinear Transforms. Recall the linear model from Chapter 3, with nonlinear transform $\mathbf{\Phi}(\mathbf{x})$ that transforms $\mathbf{x}$ to $\mathbf{z}$:

$$\mathbf{x} \rightarrow \mathbf{z} = \mathbf{\Phi}(\mathbf{x}) = [1, \phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \ldots, \phi_m(\mathbf{x})]^{\mathrm{T}}.$$

The linear model with nonlinear transform is a hypothesis of the form

$$h(\mathbf{x}) = \theta\left(w_0 + \sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right).$$

The $\phi_j(\cdot)$ are called basis functions. On face value, the neural network and the linear model look nearly identical, by setting $\theta(\mathbf{v}_j^{\mathrm{T}}\mathbf{x}) = \phi_j(\mathbf{x})$. There is a subtle difference, though, and this difference has a big practical impact. With the nonlinear transform, the basis functions $\phi_j(\cdot)$ are fixed ahead of time before you look at the data. With the neural network, the 'basis function' $\theta(\mathbf{v}_j^{\mathrm{T}}\mathbf{x})$ has a parameter $\mathbf{v}_j$ inside, and we can tune $\mathbf{v}_j$ *after* seeing the data. First, this has a computational impact because the parameter $\mathbf{v}_j$ appears *inside* the nonlinearity $\theta(\cdot)$; the model is no longer linear in its parameters. We saw a similar effect with the centers of the radial basis function network in Chapter 6. Models which are nonlinear in their parameters pose a significant computational challenge when it comes to fitting to data. Second, it means that we can *tune the basis functions to the data*. Tunable basis functions, although computationally harder to fit to data, do give us considerably more flexibility to fit the data than do fixed basis functions. With m tunable basis functions one has roughly the same approximation power to fit the data as with m^d fixed basis functions. For large d, tunable basis functions have considerably more power.

Exercise 7.10

It is no surprise that adding nodes in the hidden layer gives the neural network more approximation ability, because you are adding more parameters.

How many weight parameters are there in a neural network with architecture specified by $\mathbf{d} = [d^{(0)}, d^{(1)}, \ldots, d^{(L)}]$, a vector giving the number of nodes in each layer? Evaluate your formula for a 2 hidden layer network with 10 hidden nodes in each hidden layer.

Approximation Capability of the Neural Network. It is possible to quantify how the approximation ability of the neural network grows as you increase m, the number of hidden units. Such results fall into the field known as functional approximation theory, a field which, in the context of neural networks, has produced some interesting results. Usually one starts by making some assumption about the smoothness (or complexity) of the target function f. On the theoretical side, you have lost some generality as compared with, for example, the VC-analysis. However, in practice, such assumptions are OK because target functions are often smooth. If you assume that the data are generated by a target function with complexity[7] at most C_f, then a variety of bounds exist on how small an in-sample error is achievable with m hidden units. For regression with squared error, one can achieve in-sample error

$$E_{\text{in}}(h) = \frac{1}{N}\sum_{n=1}^{N}(h(\mathbf{x}_n) - y_n)^2 \leq \frac{(2RC_f)^2}{m},$$

where $R = \max_n \|\mathbf{x}_n\|$ is the 'radius' of the data. The in-sample error decreases inversely with the number of hidden units. For classification, a similar result with slightly worse dependence on m exists. With high probability,

$$E_{\text{in}} \leq E^*_{\text{out}} + O(C_f/\sqrt{m}),$$

where E^*_{out} is the out-of-sample error of the optimal classifier that we discussed in Chapter 6. The message is that E_{in} can be made small by choosing a large enough hidden layer.

Generalization and the VC-Dimension. For sufficiently large m, we can get E_{in} to be small, so what remains is to ensure that $E_{\text{in}} \approx E_{\text{out}}$. We need to look at the VC-dimension. For the two layer hard-threshold neural network (MLP) where $\theta(x) = \text{sign}(x)$, we show a simple bound on the VC dimension:

$$d_{\text{VC}} \leq (\text{const}) \cdot md \log(md). \tag{7.7}$$

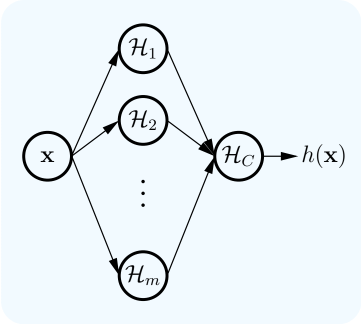
For a general sigmoid neural network, d_{VC} can be infinite. For the $\tanh(\cdot)$ sigmoid, with $\text{sign}(\cdot)$ output node, one can show that $d_{\text{VC}} = O(VQ)$ where V is the number of hidden nodes and Q is the number of weights; for the two layer case

$$d_{\text{VC}} = O(md(m + d)).$$

The $\tanh(\cdot)$ network has higher VC-dimension than the 2-layer MLP, which is not surprising because $\tanh(\cdot)$ can approximate $\text{sign}(\cdot)$ by choosing large enough weights. So every dichotomy that can be implemented by the MLP can also be implemented by the $\tanh(\cdot)$ neural network.

[7] We do not describe details of how the complexity of a target can be measured. One measure is the size of the 'high frequency' components of f in its Fourier transform. Another more restrictive measure is the number of bounded derivatives f has.

To derive (7.7), we will actually show a more general result. Consider the hypothesis set illustrated by the network on the right. Hidden node i in the hidden layer implements a function $h_i \in \mathcal{H}_i$ which maps $\mathbb{R}^d$ to $\{+1, -1\}$; the output node implements a function $h_c \in \mathcal{H}_C$ which maps $\mathbb{R}^m$ to $\{+1, -1\}$. This output node combines the outputs of the hidden layer nodes to implement the hypothesis



$$h(\mathbf{x}) = h_C(h_1(\mathbf{x}), \ldots, h_m(\mathbf{x})).$$

(For the 2-layer MLP, all the hypothesis sets are perceptrons.)

Suppose the VC-dimension of $\mathcal{H}_i$ is d_i and the VC-dimension of $\mathcal{H}_C$ is d_c. Fix $\mathbf{x}_1, \ldots, \mathbf{x}_N$, and the hypotheses $h_1, \ldots, h_m$ implemented by the hidden nodes. The hypotheses $h_1, \ldots, h_m$ are now fixed basis functions defining a transform to $\mathbb{R}^m$,

$$\mathbf{x}_1 \to \mathbf{z}_1 = \begin{bmatrix} h_1(\mathbf{x}_1) \\ \vdots \\ h_m(\mathbf{x}_1) \end{bmatrix} \quad \cdots \quad \mathbf{x}_N \to \mathbf{z}_N = \begin{bmatrix} h_1(\mathbf{x}_N) \\ \vdots \\ h_m(\mathbf{x}_N) \end{bmatrix}.$$

The transformed points are binary vectors in $\mathbb{R}^m$. Given $h_1, \ldots, h_m$, the points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ are transformed to an arrangement of points $\mathbf{z}_1, \ldots, \mathbf{z}_N$ in $\mathbb{R}^m$. Using our flexibility to choose $h_1, \ldots, h_m$, we now upper bound the number of possible *different* arrangements $\mathbf{z}_1, \ldots, \mathbf{z}_N$ we can get.

The first components of all the $\mathbf{z}_n$ are given by $h_1(\mathbf{x}_1), \ldots, h_1(\mathbf{x}_N)$, which is a dichotomy of $\mathbf{x}_1, \ldots, \mathbf{x}_N$ implemented by h_1. Since the VC-dimension of $\mathcal{H}_1$ is d_1, there are at most N^{d_1} such dichotomies.[8] That is, there are at most N^{d_1} different ways of choosing assignments to *all* the first components of the $\mathbf{z}_n$. Similarly, an assignment to all the ith components can be chosen in at most N^{d_i} ways. Thus, the total number of possible arrangements for $\mathbf{z}_1, \ldots, \mathbf{z}_N$ is at most

$$\prod_{i=1}^{m} N^{d_i} = N^{\sum_{i=1}^{m} d_i}.$$

Each of these arrangements can be dichotomized in at most N^{d_c} ways, since the VC-dimension of $\mathcal{H}_C$ is d_c. Each such dichotomy for a particular arrangement gives one dichotomy of the data $\mathbf{x}_1, \ldots, \mathbf{x}_N$. Thus, the maximum number of different dichotomies we can implement on $\mathbf{x}_1, \ldots, \mathbf{x}_N$ is upper bounded by the product: the number of possible arrangements times the number of ways

[8] Recall that for any hypothesis set with VC-dimension d_{VC} and any $N \geq d_{\text{VC}}$, $m(N)$ (the maximum number of implementable dichotomies) is bounded by $(eN/d_{\text{VC}})^{d_{\text{VC}}} \leq N^{d_{\text{VC}}}$ (for the sake of simplicity we assume that $d_{\text{VC}} \geq 2$).

of dichotomizing a particular arrangement. We have shown that

$$m(N) \leq N^{d_c} \cdot N^{\sum_{i=1}^m d_i} = N^{d_c + \sum_{i=1}^m d_i}.$$

Let $D = d_c + \sum_{i=1}^m d_i$. After some algebra (left to the reader), if $N \geq 2D \log_2 D$, then $m(N) < 2^N$, from which we conclude that $d_{\text{VC}} \leq 2D \log_2 D$. For the 2-layer MLP, $d_i = d + 1$ and $d_c = m + 1$, and so we have that $D = d_c + \sum_{i=1}^m d_i = m(d + 2) + 1 = O(md)$. Thus, $d_{\text{VC}} = O(md \log(md))$. Our analysis looks very crude, but it is almost tight: it is possible to shatter $\Omega(md)$ points with m hidden units (see Problem 7.16), and so the upper bound can be loose by at most a logarithmic factor. Using the VC-dimension, the generalization error bar from Chapter 2 is $O(\sqrt{(d_{\text{VC}} \log N)/N})$ which for the 2-layer MLP is $O(\sqrt{(md \log(md) \log N)/N})$.

We will get good generalization if m is not too large and we can fit the data if m is large enough. A balance is called for. For example, choosing $m = \frac{1}{d}\sqrt{N}$ as $N \to \infty$, $E_{\text{out}} \to E_{\text{in}}$ and $E_{\text{in}} \to E_{\text{out}}^*$. That is, $E_{\text{out}} \to E_{\text{out}}^*$ (the optimal performance) as N grows, and m grows sub-linearly with N. In practice the 'asymptotic' regime is a luxury and one does not simply set $m \approx \sqrt{N}$. These theoretical results are a good guideline, but the best out-of-sample performance usually results when you control overfitting using validation (to select the number of hidden units) and regularization to prevent overfitting.

We conclude with a note on where neural networks sit in the parametric-nonparametric debate. There are explicit parameters to be learned, so parametric seems right. But distinctive features of nonparametric models also stand out: the neural network is generic and flexible and can realize optimal performance when N grows. Neither parametric nor nonparametric captures the whole story. We choose to label neural networks as *semi-parametric*.

7.4 Regularization and Validation

The multi-layer neural network is powerful, and, coupled with gradient descent (a good algorithm to minimize E_{in}), we have a recipe for overfitting. We discuss some practical techniques to help.

7.4.1 Weight Based Complexity Penalties

As with linear models, one can regularize the learning using a complexity penalty by minimizing an augmented error (penalized in-sample error). The squared weight decay regularizer is popular, having augmented error:

$$E_{\text{aug}}(\mathbf{w}) = E_{\text{in}}(\mathbf{w}) + \frac{\lambda}{N} \sum_{\ell,i,j} (w_{ij}^{(\ell)})^2$$

The regularization parameter λ is selected via validation, as discussed in Chapter 4. To apply gradient descent, we need $\nabla E_{\text{aug}}(\mathbf{w})$. The penalty term adds

to the gradient a term proportional to weights,

$$\frac{\partial E_{\text{aug}}(\mathbf{w})}{\partial \mathrm{W}^{(\ell)}} = \frac{\partial E_{\text{in}}(\mathbf{w})}{\partial \mathrm{W}^{(\ell)}} + \frac{2\lambda}{N}\mathrm{W}^{(\ell)}.$$

We know how to obtain $\partial E_{\text{in}}/\partial \mathrm{W}^{(\ell)}$ using backpropagation. The penalty term adds a component to the weight update that is in the negative direction of $\mathbf{w}$, i.e. towards zero weights – hence the term weight decay.

Another similar regularizer is *weight elimination*, having augmented error:

$$E_{\text{aug}}(\mathbf{w}, \lambda) = E_{\text{in}}(\mathbf{w}) + \frac{\lambda}{N}\sum_{\ell,i,j}\frac{(w_{ij}^{(\ell)})^2}{1+(w_{ij}^{(\ell)})^2}.$$

For a small weight, the penalty term is much like weight decay, and will decay that weight to zero. For a large weight, the penalty term is approximately a constant, and contributes little to the gradient. Small weights decay faster than large weights, and the effect is to 'eliminate' those smaller weights.

Exercise 7.11

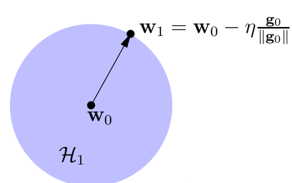
For weight elimination, show that $\dfrac{\partial E_{\text{aug}}}{\partial w_{ij}^{(\ell)}} = \dfrac{\partial E_{\text{in}}}{\partial w_{ij}^{(\ell)}} + \dfrac{2\lambda}{N}\cdot\dfrac{w_{ij}^{(\ell)}}{(1+(w_{ij}^{(\ell)})^2)^2}$.

Argue that weight elimination shrinks small weights faster than large ones.

7.4.2 Early Stopping

Another method for regularization, which on face value does not look like regularization is early stopping. An iterative method such as gradient descent does not explore your full hypothesis set all at once. With more iterations, more of your hypothesis set is explored. This means that by using fewer iterations, you explore a smaller hypothesis set and should get better generalization.[9]

Consider fixed-step gradient descent with step size η. At the first step, we start at weights $\mathbf{w}_0$, and take a step of size η to $\mathbf{w}_1 = \mathbf{w}_0 - \eta\frac{\mathbf{g}_0}{\|\mathbf{g}_0\|}$. Because we have taken a step in the direction of the negative gradient, we have 'looked at' all the hypotheses in the shaded region shown on the right.



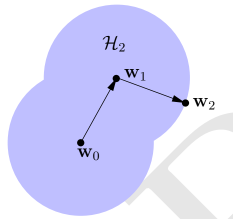
This is because a step in the negative gradient leads to the sharpest decrease in $E_{\text{in}}(\mathbf{w})$, and so $\mathbf{w}_1$ minimizes $E_{\text{in}}(\mathbf{w})$ among all weights with $\|\mathbf{w}-\mathbf{w}_0\| \le \eta$. We indirectly searched the entire hypothesis set

$$\mathcal{H}_1 = \{\mathbf{w} : \ \|\mathbf{w}-\mathbf{w}_0\| \le \eta\},$$

and picked the hypothesis $\mathbf{w}_1 \in \mathcal{H}_1$ with minimum in-sample error.

[9] If we are to be sticklers for correctness, the hypothesis set explored could depend on the data set and so we cannot directly apply the VC analysis which requires the hypothesis set to be fixed ahead of time. Since we are just illustrating the main idea, we will brush such technicalities under the rug.

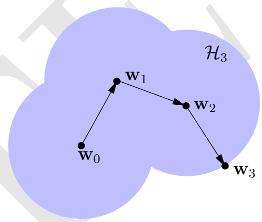
Now consider the second step, as illustrated to the right, which moves to $\mathbf{w}_2$. We indirectly explored the hypothesis set of weights with $\|\mathbf{w} - \mathbf{w}_1\| \leq \eta$, picking the best. Since $\mathbf{w}_1$ was already the minimizer of E_{in} over $\mathcal{H}_0$, this means that $\mathbf{w}_2$ is the minimizer of E_{in} among all hypotheses in $\mathcal{H}_2$, where



$$\mathcal{H}_2 = \mathcal{H}_1 \cup \{\mathbf{w} : \ \|\mathbf{w} - \mathbf{w}_1\| \leq \eta\}.$$

Note that $\mathcal{H}_1 \subset \mathcal{H}_2$. Similarly, we define hypothesis set

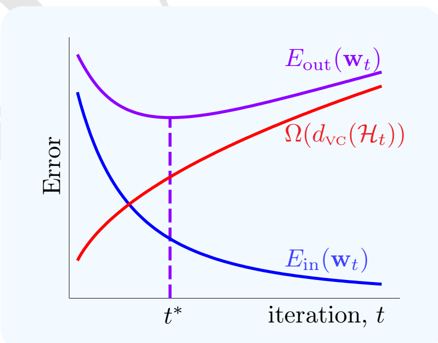
$$\mathcal{H}_3 = \mathcal{H}_2 \cup \{\mathbf{w} : \ \|\mathbf{w} - \mathbf{w}_2\| \leq \eta\},$$



and in the 3rd iteration, we pick weights $\mathbf{w}_3$ than minimize E_{in} over $\mathbf{w} \in \mathcal{H}_3$. We can continue this argument as gradient descent proceeds, and define a nested sequence of hypothesis sets

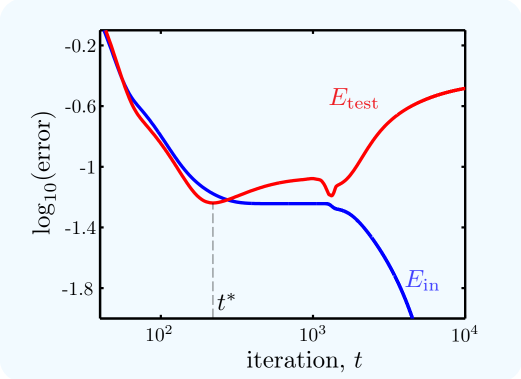
$$\mathcal{H}_1 \subset \mathcal{H}_2 \subset \mathcal{H}_3 \subset \mathcal{H}_4 \subset \cdots.$$

As t increases, $E_{\text{in}}(\mathbf{w}_t)$ is decreasing, and $d_{\text{VC}}(\mathcal{H}_t)$ is increasing. So, we expect to see the approximation-generalization trade-off which was illustrated in Figure 2.3 (reproduced here with iteration t a proxy for d_{VC}):



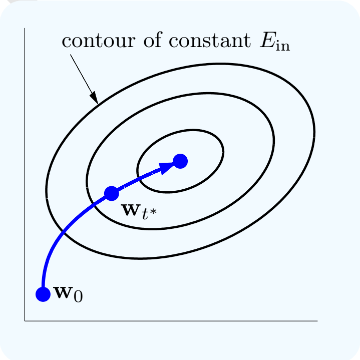
The figure suggests it may be better to stop early at some t^*, well before reaching a minimum of E_{in}. Indeed, this picture is observed in practice.

Example 7.2. We revisit the digits task of classifying '1' versus all other digits, with 70 randomly selected data points and a small sigmoidal neural network with a single hidden unit and $\tanh(\cdot)$ output node. The figure below shows the in-sample error and test error versus iteration number.



The curves reinforce our theoretical discussion: the test error initially decreases as the approximation gain overcomes the worse generalization error bar; then, the test error increases as the generalization error bar begins to dominate the approximation gain, and overfitting becomes a serious problem. □

In the previous example, despite using a parsimonious neural network with just a single hidden node, overfitting was an issue because the data are noisy and the target function is complex, so both stochastic and deterministic noise are significant. We need to regularize.



In the example, it is better to stop early at t^* and *constrain* the learning to the smaller hypothesis set $\mathcal{H}_{t^*}$. In this sense, early stopping is a form of regularization. Early stopping is related to weight decay, as illustrated to the right. You initialize $\mathbf{w}_0$ near zero; if you stop early at $\mathbf{w}_{t^*}$ you have stopped at weights closer to $\mathbf{w}_0$, i.e., smaller weights. Early stopping indirectly achieves smaller weights, which is what weight decay directly achieves. To determine when to stop training, use a validation set to monitor the validation error at iteration t as you minimize the training-set error. Report the weights $\mathbf{w}_{t^*}$ that have minimum validation error when you are done training.

After selecting t^*, it is tempting to use all the data to train for t^* iterations. Unfortunately, adding back the validation data and training for t^* iterations can lead to a completely different set of weights. The validation estimate of performance only holds for $\mathbf{w}_{t^*}$ (the weights you should output). This appears to go against the wisdom of the decreasing learning curve from Chapter 4: if you learn with more data, you get a better final hypothesis.[10]

[10] Using all the data to train to an in-sample error of $E_{\text{train}}(w_{t^*})$ is also not recommended. Further, an in-sample error of $E_{\text{train}}(w_{t^*})$ may not even be achievable with all the data.

Exercise 7.12

Why does outputting $\mathbf{w}_{t^*}$ rather than training with all the data for t^* iterations *not* go against the wisdom that learning with more data is better.

[Hint: "More data is better" applies to a **fixed** *model* $(\mathcal{H}, \mathcal{A})$*. Early stopping is model selection on a nested hypothesis sets* $\mathcal{H}_1 \subset \mathcal{H}_2 \subset \cdots$ *determined by* $\mathcal{D}_{\text{train}}$*. What happens if you were to use the full data* $\mathcal{D}$*?]*

When using early stopping, the usual trade-off exists for choosing the size of the validation set: too large and there is little data to train on; too small and the validation error will not be reliable.A rule of thumb is to set aside a fraction of the data (one-tenth to one-fifth) for validation.

Exercise 7.13

Suppose you run gradient descent for 1000 iterations. You have 500 examples in $\mathcal{D}$, and you use 450 for $\mathcal{D}_{\text{train}}$ and 50 for $\mathcal{D}_{\text{val}}$. You output the weight from iteration 50, with $E_{\text{val}}(\mathbf{w}_{50}) = 0.05$ and $E_{\text{train}}(\mathbf{w}_{50}) = 0.04$.

(a) Is $E_{\text{val}}(\mathbf{w}_{50}) = 0.05$ an unbiased estimate of $E_{\text{out}}(\mathbf{w}_{50})$?

(b) Use the Hoeffding bound to get a bound for E_{out} using E_{val} plus an error bar. Your bound should hold with probability at least 0.1.

(c) Can you bound E_{out} using E_{train} or do you need more information?

Example 7.2 also illustrates another common problem with the sigmoidal output node: gradient descent often hits a flat region where E_{in} decreases very little.[11] You might stop training, thinking you found a local minimum. This 'early stopping' by mistake is sometimes called the 'self-regularizing' property of sigmoidal neural networks. Accidental regularization due to misinterpreted convergence is unreliable. Validation is much better.

7.4.3 Experiments With Digits Data

Let's put theory to practice on the digits task (to classify '1' versus all other digits). We learn on 500 randomly chosen data points using a sigmoidal neural network with one hidden layer and 10 hidden nodes. There are 41 weights (tunable parameters), so more than 10 examples per degree of freedom, which is quite reasonable. We use identity output transformation $\theta(s) = s$ to reduce the possibility of getting stuck at a flat region of the error surface. At the end of training, we use the output transformation $\theta(s) = \text{sign}(s)$ for actually classifying data. After more than *2 million* iterations of gradient descent, we manage to get close to a local minimum. The result is shown in Figure 7.2. It doesn't take a genius to see the overfitting. Figure 7.2 attests to the approximation capabilities of a moderately sized neural network. Let's try weight

[11] The linear output transformation function helps avoid such excessively flat regions.

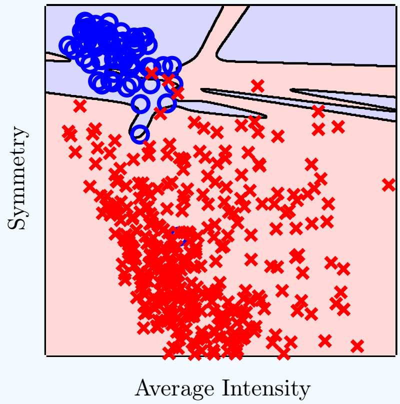
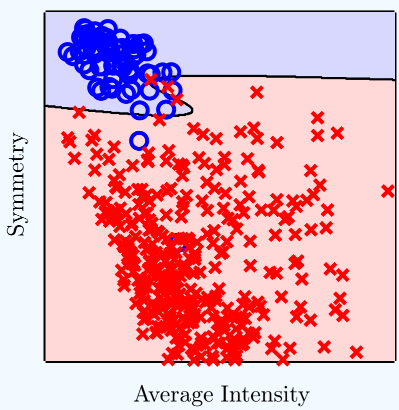
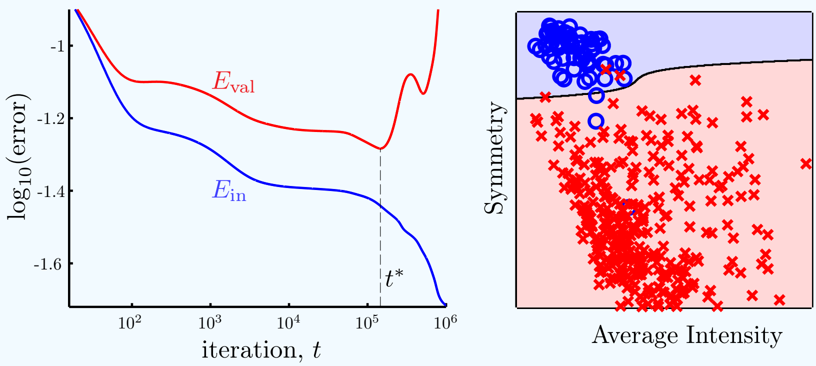


Figure 7.2: 10 hidden unit neural network trained with gradient descent on 500 examples from the digits data (no regularization). Blue circles are the digit '1' and the red x's are the other digits. Overfitting is rampant.

decay to fight the overfitting. We minimize $E_{\text{aug}}(\mathbf{w}, \lambda) = E_{\text{in}}(\mathbf{w}) + \frac{\lambda}{N}\mathbf{w}^{\text{T}}\mathbf{w}$, with $\lambda = 0.01$. We get a much more believable separator, shown below.



As a final illustration, let's try early stopping with a validation set of size 50 (one-tenth of the data); so the training set will now have size 450.
The training dynamics of gradient descent are shown in Figure 7.3(a). The linear output transformation function has helped as there are no extremely flat periods in the training error. The classification boundary with early stopping at t^* is shown in Figure 7.3(b). The result is similar to weight decay. In both cases, the regularized classification boundary is more believable. Ultimately, the quantitative statistics are what matters, and these are summarized below.



(a) Training dynamics (b) Final hypothesis

Figure 7.3: Early stopping with 500 examples from the digits data. (a) Training and validation errors for gradient descent with a training set of size 450 and validation set of size 50. (b) The 'regularized' final hypothesis obtained by early stopping at t^*, the minimum validation error.

	E_{train}	E_{val}	E_{in}	E_{out}
No Regularization	–	–	0.2%	3.1%
Weight Decay	–	–	1.0%	2.1%
Early Stopping	1.1%	2.0%	1.2%	2.0%

7.5 Beefing Up Gradient Descent

Gradient descent is a simple method to minimize E_{in} that has problems converging, especially with flat error surfaces. One solution is to minimize a friendlier error instead, which is why training with a linear output node helps. Rather than change the error measure, there is plenty of room to improve the algorithm itself. Gradient descent takes a step of size η in the negative gradient direction. How should we determine η and is the negative gradient the best direction in which to move?

Exercise 7.14

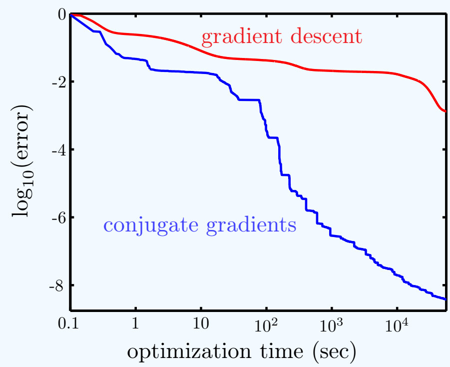
Consider the error function $E(\mathbf{w}) = (\mathbf{w} - \mathbf{w}^*)^{\mathrm{T}}\mathrm{Q}(\mathbf{w} - \mathbf{w}^*)$, where Q is an arbitrary positive definite matrix. Set $\mathbf{w} = \mathbf{0}$.

Show that the gradient $\nabla E(\mathbf{w}) = -\mathrm{Q}\mathbf{w}^*$. What weights minimize $E(\mathbf{w})$. Does gradient descent move you in the direction of these optimal weights?

Reconcile your answer with the claim in Chapter 3 that the gradient is the best direction in which to take a step. *[Hint: How big was the step?]*

The previous exercise shows that the negative gradient is not necessarily the best direction for a large step, and we would like to take larger steps to increase

the efficiency of the optimization. The next figure shows two algorithms: our old friend gradient descent and our soon-to-be friend conjugate gradient descent. Both algorithms are minimizing E_{in} for a 5 hidden unit neural network fitting 200 digits data. The performance difference is dramatic.

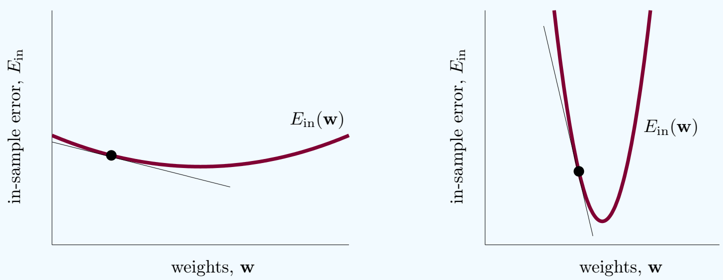


We now discuss methods for 'beefing up' gradient descent, but only scratch the surface of this important topic known as numerical optimization. The two main steps in an iterative optimization procedure are to determine:

1. Which direction should one search for a local optimum?
2. How large a step should one take in that direction?

7.5.1 Choosing the Learning Rate η

In gradient descent, the learning rate η multiplies the negative gradient to give the move $-\eta\nabla E_{\text{in}}$. The size of the step taken is proportional to η. The optimal step size (and hence learning rate η) depends on how wide or narrow the error surface is near the minimum.



wide: use large η. narrow: use small η.

When the surface is wider, we can take larger steps without overshooting; since $\|\nabla E_{\text{in}}\|$ is small, we need a large η. Since we do not know ahead of time how wide the surface is, it is easy to choose an inefficient value for η.

Variable learning rate gradient descent. A simple heuristic that adapts the learning rate to the error surface works well in practice. If the error drops, increase η; if not, the step was too large, so reject the update and decrease η. For little extra effort, we get a significant boost to gradient descent.

Variable Learning Rate Gradient Descent:

Initialize $\mathbf{w}(0)$, and η_0 at $t = 0$. Set $\alpha > 1$ and $\beta < 1$.
while stopping criterion has not been met **do**
  Let $\mathbf{g}(t) = \nabla E_{\text{in}}(\mathbf{w}(t))$, and set $\mathbf{v}(t) = -\mathbf{g}(t)$.
  if $E_{\text{in}}(\mathbf{w}(t) + \eta_t \mathbf{v}(t)) < E_{\text{in}}(\mathbf{w}(t))$ **then**
    accept: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta_t \mathbf{v}(t)$; $\eta_{t+1} = \alpha\eta_t$.
  else
    reject: $\mathbf{w}(t+1) = \mathbf{w}(t)$; $\eta_{t+1} = \beta\eta_t$.
  Iterate to the next step, $t \leftarrow t + 1$.

It is usually best to go with a conservative increment parameter, for example $\alpha \approx 1.05 - 1.1$, and a bit more aggressive decrement parameter, for example $\beta \approx 0.5 - 0.8$. This is because, if the error doesn't drop, then one is in an unusual situation and more drastic action is called for.

After a little thought, one might wonder why we need a learning rate at all. Once the direction in which to move, $\mathbf{v}(t)$, has been determined, why not simply continue along that direction until the error stops decreasing? This leads us to *steepest descent* – gradient descent with *line search*.

Steepest Descent. Gradient descent picks a descent direction $\mathbf{v}(t) = -\mathbf{g}(t)$ and updates the weights to $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta\mathbf{v}(t)$. Rather than pick η arbitrarily, we will choose the optimal η that minimizes $E_{\text{in}}(\mathbf{w}(t+1))$. Once you have the direction to move, make the best of it by moving along the line $\mathbf{w}(t) + \eta\mathbf{v}(t)$ and stopping when E_{in} is minimum (hence the term line search). That is, choose a step size η^*, where

$$\eta^*(t) = \underset{\eta}{\operatorname{argmin}}\, E_{\text{in}}(\mathbf{w}(t) + \eta\mathbf{v}(t)).$$

Steepest Descent (Gradient Descent + Line Search):

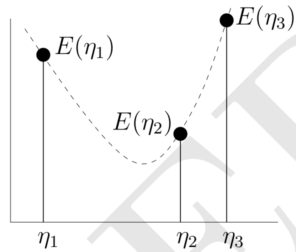
1: Initialize $\mathbf{w}(0)$ and set $t = 0$;
2: **while** stopping criterion has not been met **do**
3:   Let $\mathbf{g}(t) = \nabla E_{\text{in}}(\mathbf{w}(t))$, and set $\mathbf{v}(t) = -\mathbf{g}(t)$.
4:   Let $\eta^* = \operatorname{argmin}_\eta E_{\text{in}}(\mathbf{w}(t) + \eta\mathbf{v}(t))$.
5:   $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta^*\mathbf{v}(t)$.
6:   Iterate to the next step, $t \leftarrow t + 1$.

The line search in step 4 is a one dimensional optimization problem. Line search is an important step in most optimization algorithms, so an efficient algorithm is called for. Write $E(\eta)$ for $E_{\text{in}}(\mathbf{w}(t) + \eta\mathbf{v}(t))$. The goal is to find a minimum of $E(\eta)$. We give a simple algorithm based on binary search.

Line Search. The idea is to find an interval on the line which is guaranteed to contain a local minimum. Then, rapidly narrow the size of this interval while maintaining as an invariant the fact that it contains a local minimum.

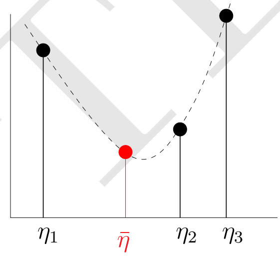
The basic invariant is a U-arrangement:

$$\eta_1 < \eta_2 < \eta_3 \quad \text{with}$$
$$E(\eta_2) < \min\{E(\eta_1), E(\eta_3)\}.$$



Since E is continuous, there must be a local minimum in the interval $[\eta_1, \eta_3]$. Now, consider the midpoint of the interval,

$$\bar{\eta} = \frac{1}{2}(\eta_1 + \eta_3),$$



hence the name *bisection algorithm*. Suppose that $\bar{\eta} < \eta_2$ as shown. If $E(\bar{\eta}) < E(\eta_2)$ then $\{\eta_1, \bar{\eta}, \eta_2\}$ is a new, smaller U-arrangement; and, if $E(\bar{\eta}) > E(\eta_2)$, then $\{\bar{\eta}, \eta_2, \eta_3\}$ is the new smaller U-arrangement. In either case, the bisection process can be iterated with the new U-arrangement. If $\bar{\eta}$ happens to equal η_2, perturb $\bar{\eta}$ slightly to resolve the degeneracy. We leave it to the reader to determine how to obtain the new smaller U-arrangement for the case $\bar{\eta} > \eta_2$.

An efficient algorithm to find an initial U-arrangement is to start with $\eta_1 = 0$ and $\eta_2 = \epsilon$ for some step ϵ. If $E(\eta_2) < E(\eta_1)$, consider the sequence

$$\eta = 0, \epsilon, 2\epsilon, 4\epsilon, 8\epsilon, \ldots$$

(each time the step doubles). At some point, the error must increase. When the error increases for the first time, the last three steps give a U-arrangement. If, instead, $E(\eta_1) < E(\eta_2)$, consider the sequence

$$\eta = \epsilon, 0, -\epsilon, -2\epsilon, -4\epsilon, -8\epsilon, \ldots$$

(the step keeps doubling but in the reverse direction). Again, when the error increases for the first time, the last three steps give a U-arrangement.[12]

Exercise 7.15

Show that $|\eta_3 - \eta_1]$ decreases exponentially in the bisection algorithm. *[Hint: show that two iterations at least halve the interval size.]*

The bisection algorithm continues to bisect the interval and update to a new U-arrangement until the length of the interval $|\eta_3 - \eta_1|$ is small enough, at which

[12] We do not worry about $E(\eta_1) = E(\eta_2)$ – such ties can be broken by small perturbations.

point you can return the midpoint of the interval as the approximate local minimum. Usually 20 iterations of bisection are enough to get an acceptable solution. A better quadratic interpolation algorithm is given in Problem 7.8, which only needs about 4 iterations in practice.

Example 7.3. We illustrate these three heuristics for improving gradient descent on our digit recognition (classifying '1' versus other digits). We use 200 data points and a neural network with 5 hidden units. We show the performance of gradient descent, gradient descent with variable learning rate, and steepest descent (line search) in Figure 7.4. The table below summarizes the in-sample error at various points in the optimization.

	Optimization Time		
Method	10 sec	1,000 sec	50,000 sec
Gradient Descent	0.079	0.0206	0.00144
Stochastic Gradient Descent	**0.0213**	**0.00278**	0.000022
Variable Learning Rate	0.039	0.014	0.00010
Steepest Descent	0.043	0.0189	**0.000012**

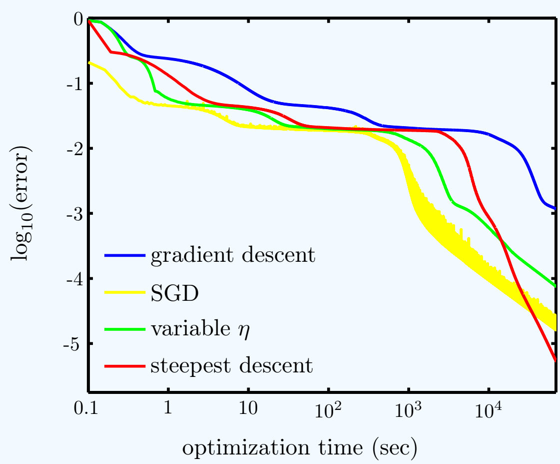
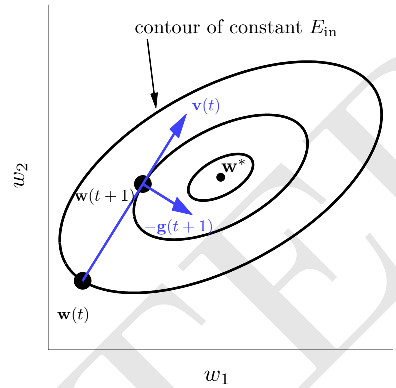


Figure 7.4: Gradient descent, variable learning rate and steepest descent using digits data and a 5 hidden unit 2-layer neural network with linear output. For variable learning rate, $\alpha = 1.1$ and $\beta = 0.8$.

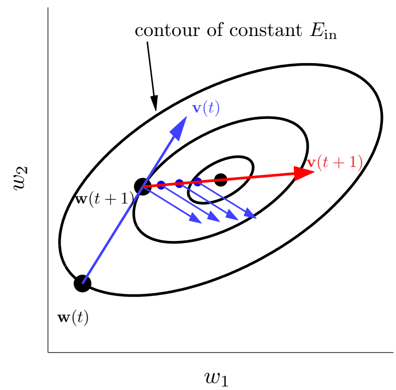
Note that SGD is quite competitive. The figure illustrates why it is hard to know when to stop minimizing. A flat region 'trapped' all the methods, even though we used a linear output node transform. It is very hard to differentiate between a flat region (which is typically caused by a very steep valley that leads to inefficient zig-zag behavior) and a true local minimum. □

7.5.2 Conjugate Gradient Minimization

Conjugate gradient is a queen among optimization methods because it leverages a simple principle. Don't undo what you have already accomplished. When you end a line search, because the error cannot be further decreased by moving back or forth along the search direction, it must be that the new gradient and the previous line search direction are orthogonal. What this means is that you have succeeded in setting one of the components of the gradient to zero, namely the component along the search direction $\mathbf{v}(t)$ (see the figure). If the next search direction is the negative of the new gradient, it will be orthogonal to the previous search direction.



You are at a local minimum when the gradient is zero, and setting one component to zero is certainly a step in the right direction. As you move along the next search direction (for example the new negative gradient), the gradient will change and may not remain orthogonal to the previous search direction, a task you laboriously accomplished in the previous line search. The conjugate gradient algorithm chooses the next direction $\mathbf{v}(t+1)$ so that the gradient along this direction, *will remain perpendicular to the previous search direction* $\mathbf{v}(t)$. This is called the conjugate direction, hence the name. After a line search along this new direction $\mathbf{v}(t+1)$ to minimize E_{in}, you will have set *two* components of the gradient to zero. First, the gradient remained perpendicular to the previous search direction $\mathbf{v}(t)$. Second, the gradient will be orthogonal to $\mathbf{v}(t+1)$ because of the line search (see the figure). The gradient along the new direction $\mathbf{v}(t+1)$ is shown by the blue arrows in the figure. Because $\mathbf{v}(t+1)$ is conjugate to $\mathbf{v}(t)$, observe how the gradient as we move along $\mathbf{v}(t+1)$ remains orthogonal to the previous direction $\mathbf{v}(t)$.



Exercise 7.16

Why does the new search direction pass through the optimal weights?

We made progress! Now two components of the gradient are zero. In two dimensions, this means that the gradient itself must be zero and we are done.

In higher dimension, if we could continue to set a component of the gradient to zero with each line search, maintaining all previous components at zero, we will eventually set every component of the gradient to zero and be at a local minimum. Our discussion is true for an idealized quadratic error function. In general, conjugate gradient minimization implements our idealized expectations approximately. Nevertheless, it works like a charm because the idealized setting is a good approximation once you get close to a local minimum, and this is where algorithms like gradient descent become ineffective.

Now for the details. The algorithm constructs the current search direction as a linear combination of the previous search direction and the current gradient,

$$\mathbf{v}(t) = -\mathbf{g}(t) + \mu_t \cdot \mathbf{v}(t-1),$$

where

$$\mu_t = \frac{\mathbf{g}(t+1)^{\mathrm{T}}(\mathbf{g}(t+1) - \mathbf{g}(t))}{\mathbf{g}(t)^{\mathrm{T}}\mathbf{g}(t)}.$$

The term $\mu_t \cdot \mathbf{v}(t-1)$ is called the *momentum* term because it asks you to keep moving in the same direction you were moving in. The multiplier μ_t is called the momentum parameter. The full conjugate gradient descent algorithm is summarized in the following algorithm box.

Conjugate Gradient Descent:

1: Initialize $\mathbf{w}(0)$ and set $t = 0$; set $\mathbf{v}(-1) = \mathbf{0}$
2: **while** stopping criterion has not been met **do**
3: Let $\mathbf{v}(t) = -\mathbf{g}(t) + \mu_t\mathbf{v}(t-1)$, where

$$\mu_t = \frac{\mathbf{g}(t+1)^{\mathrm{T}}(\mathbf{g}(t+1) - \mathbf{g}(t))}{\mathbf{g}(t)^{\mathrm{T}}\mathbf{g}(t)}.$$

4: Let $\eta^* = \operatorname{argmin}_\eta E_{\text{in}}(\mathbf{w}(t) + \eta\mathbf{v}(t))$.
5: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta^*\mathbf{v}(t)$;
6: Iterate to the next step, $t \leftarrow t+1$;

The only difference between conjugate gradient descent and steepest descent is in step 3 where the line search direction is different from the negative gradient. Contrary to intuition, the negative gradient direction is not always the best direction to move, because it can undo some of the good work you did before.

In practice, for error surfaces that are not exactly quadratic, the $\mathbf{v}(t)$'s are only approximately conjugate and it is recommended that you 'restart' the algorithm by setting μ_t to zero every so often (for example every d iterations). That is, every d iterations you throw in a steepest descent iteration.

Example 7.4. Continuing with the digits example, we compare conjugate gradient and the previous champion steepest descent in the next table and Figure 7.5.

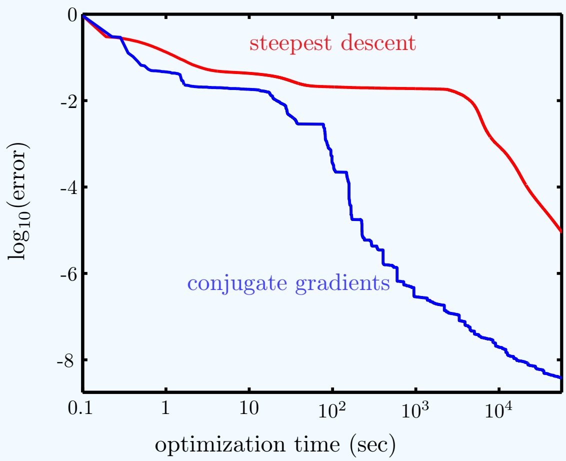


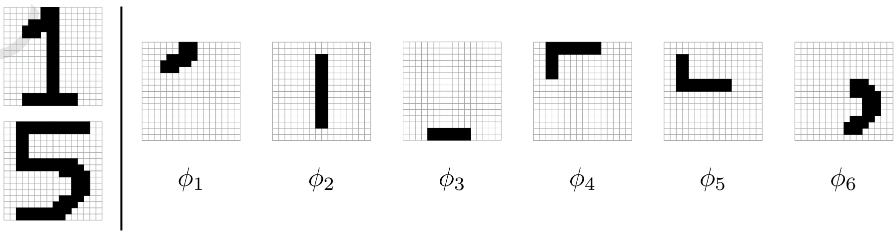
Figure 7.5: Steepest descent versus conjugate gradient descent using 200 examples of the digits data and a 2-layer sigmoidal neural network with 5 hidden units.

Method	Optimization Time		
	10 sec	1,000 sec	50,000 sec
Steepest Descent	0.043	0.0189	1.2×10^{-5}
Conjugate Gradients	**0.0200**	$\mathbf{1.13 \times 10^{-6}}$	$\mathbf{2.73 \times 10^{-9}}$

The performance difference is dramatic. □

7.6 Deep Learning: Networks with Many Layers

Universal approximation says that a single hidden layer with enough hidden units can approximate any target function. But, that may not be a natural way to represent the target function. Often, many layers more closely mimics human learning. Let's get our feet wet with the digit recognition problem to classify '1' versus '5'. A natural first step is to decompose the two digits into basic components, just as one might break down a face into two eyes, a nose, a mouth, two ears, etc. Here is one attempt for a prototypical '1' and '5'.

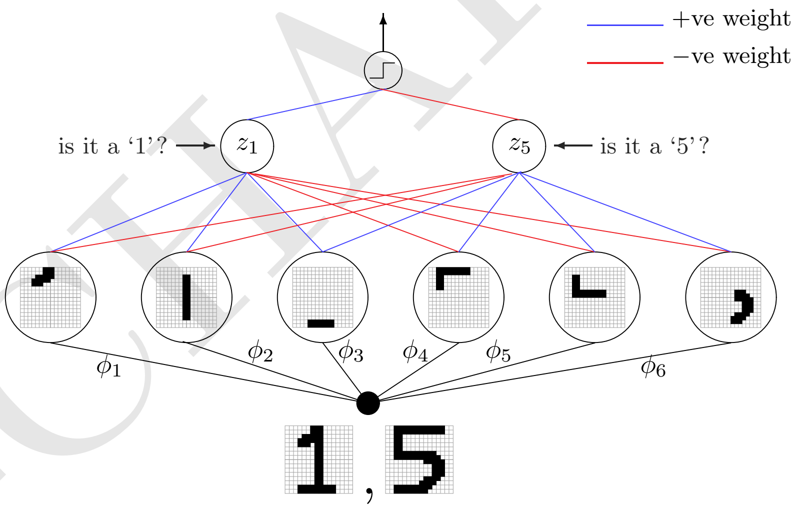


Indeed, we could plausibly argue that every '1' should contain a ϕ_1, ϕ_2 and ϕ_3; and, every '5' should contain a ϕ_3, ϕ_4, ϕ_5, ϕ_6 and perhaps a little ϕ_1. We have deliberately used the notation ϕ_i which we used earlier for the coordinates of the feature transform Φ. These basic shapes are *features* of the input, and, for example, we would like ϕ_1 to be large (close to 1) if its corresponding feature is in the input image and small (close to -1) if not.

Exercise 7.17

The basic shape ϕ_3 is in both the '1' and the '5'. What other digits do you expect to contain each basic shape $\phi_1 \cdots \phi_6$. How would you select additional basic shapes if you wanted to distinguish between all the digits. (What properties should useful basic shapes satisfy?)

We can build a classifier for '1' versus '5' from these basic shapes. Remember how, at the beginning of the chapter, we built a complex Boolean function from the 'basic' functions AND and OR? Let's mimic that process here. The complex function we are building is the digit classifier and the basic functions are our features. Assume, for now, that we have feature functions ϕ_i which compute the presence ($+1$) or absence (-1) of the corresponding feature. Take a close look at the following network and work it through from input to output.



Ignoring details like the exact values of the weights, node z_1 answers the question "is the image a '1'?" and similarly node z_5 answers "is the image a '5'?" Let's see why. If they have done their job correctly when we feed in a '1', ϕ_1, ϕ_2, ϕ_3 compute $+1$, and ϕ_4, ϕ_5, ϕ_6 compute -1. Combining $\phi_1, \ldots, \phi_6$ with the signs of the weights on outgoing edges, all the inputs to z_1 will be positive hence z_1 outputs $+1$; all but one of the inputs into z_5 are negative, hence z_5 outputs -1. A similar analysis holds if you feed in the '5'. The final

node combines z_1 and z_5 to the final output. At this point, it is useful to fill in all the blanks with an exercise.

Exercise 7.18

Since the input $\mathbf{x}$ is an image it is convenient to represent it as a matrix $[x_{ij}]$ of its pixels which are black ($x_{ij} = 1$) or white ($x_{ij} = 0$). The basic shape ϕ_k identifies a set of these pixels which are black.

(a) Show that feature ϕ_k can be computed by the neural network node

$$\phi_k(\mathbf{x}) = \tanh\left(w_0 + \sum_{ij} w_{ij} x_{ij}\right).$$

(b) What are the inputs to the neural network node?

(c) What do you choose as values for the weights? *[Hint: consider separately the weights of the pixels for those $x_{ij} \in \phi_k$ and those $x_{ij} \notin \phi_k$.]*

(d) How would you choose w_0? (Not all digits are written identically, and so a basic shape may not always be exactly represented in the image.)

(e) Draw the final network, filling in as many details as you can.

You may have noticed, that the output of z_1 is all we need to solve our problem. This would not be the case if we were solving the full multi-class problem with nodes $z_0, \ldots, z_9$ corresponding to all ten digits. Also, we solved our problem with relative ease – our 'deep' network has just 2 hidden layers. In a more complex problem, like face recognition, the process would start just as we did here, with basic shapes. At the next level, we would constitute more complicated shapes from the basic shapes, but we would not be home yet. These more complicated shapes would constitute still more complicated shapes until at last we had realistic objects like eyes, a mouth, ears, etc. There would be a hierarchy of 'basic' features until we solve our problem at the very end.

Now for the punch line and crux of our story. The punch line first. Shine your floodlights back on the network we constructed, and scrutinize what the different layers are doing. The first layer constructs a low-level representation of basic shapes; the next layer builds a higher level representation from these basic shapes. As we progress up more layers, we get more complex representations in terms of simpler parts from the previous layer: an 'intelligent' decomposition of the problem, starting from simple and getting more complex, until finally the problem is solved. This is the promise of the deep network, that it provides some human-like insight into how the problem is being solved based on a hierarchy of more complex representations for the input. While we might attain a solution of similar accuracy with a single hidden layer, we would gain no such insight. The picture is rosy for our intuitive digit recognition problem, but here is the crux of the matter: for a complex learning problem, how do we automate all of this in a computer algorithm?

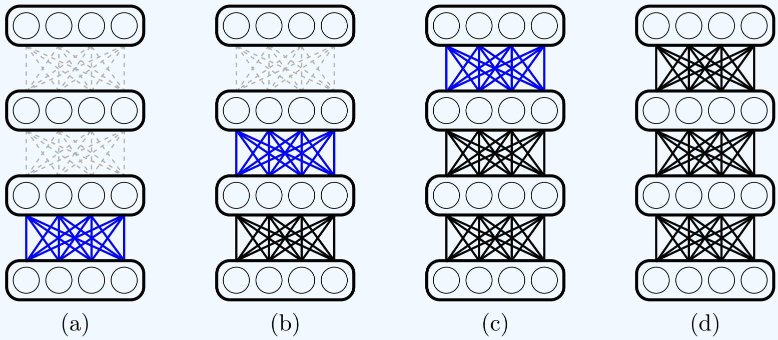


Figure 7.6: Greedy deep learning algorithm. (a) First layer weights are learned. (b) First layer is fixed and second layer weights are learned. (c) First two layers are fixed and third layer weights are learned. (d) Learned weights can be used as a starting point to fine-tune the entire network.

7.6.1 A Greedy Deep Learning Algorithm

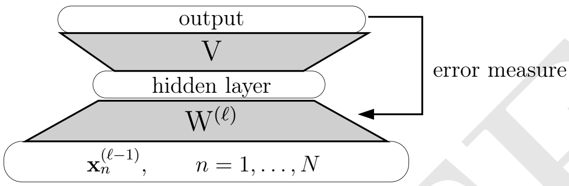
Historically, the shallow (single hidden layer) neural network was favored over the deep network because deep networks are hard to train, suffer from many local minima and, relative to the number of tunable parameters, they have a very large tendency to overfit (composition of nonlinearities is typically much more powerful than a linear combination of nonlinearities). Recently, some simple heuristics have shown good performance empirically and have brought deep networks back into the limelight. Indeed, the current best algorithm for digit recognition is a deep neural network trained with such heuristics.

The greedy heuristic has a general form. Learn the first layer weights $\mathrm{W}^{(1)}$ and fix them.[13] The output of the first hidden layer is a nonlinear transformation of the inputs $\mathbf{x}_n \to \mathbf{x}_n^{(1)}$. These outputs $\mathbf{x}_n^{(1)}$ are used to train the second layer weights $\mathrm{W}^{(2)}$, *while keeping the first layer weights fixed*. This is the essence of the greedy algorithm, to 'greedily' pick the first layer weights, fix them, and then move on to the second layer weights. One ignores the possibility that better first layer weights might exist if one takes into account what the second layer is doing. The process continues with the outputs $\mathbf{x}^{(2)}$ used to learn the weights $\mathrm{W}^{(3)}$, and so on.

[13]Recall that we use the superscript $(\cdot)^{(\ell)}$ to denote the layer ℓ.

Greedy Deep Learning Algorithm:

1: **for** $\ell = 1, \dots, L$ **do**
2: $\quad \mathrm{W}^{(1)} \cdots \mathrm{W}^{(\ell-1)}$ are given from previous iterations.
3: $\quad$ Compute layer $\ell - 1$ outputs $\mathbf{x}_n^{(\ell-1)}$ for $n = 1, \dots, N$.
4: $\quad$ Use $\{\mathbf{x}_n^{(\ell-1)}\}$ to learn weights W^{ℓ} by training a *single* hidden layer neural network. ($\mathrm{W}^{(1)} \cdots \mathrm{W}^{(\ell-1)}$ are fixed.)



We have to clarify step 4 in the algorithm. The weights $\mathrm{W}^{(\ell)}$ and V are learned, though V is not needed in the algorithm. To learn the weights, we minimize an error (which will depend on the output of the network), and that error is not yet defined. To define the error, we must first define the output and then how to compute the error from the output.

Unsupervised Auto-encoder. One approach is to take to heart the notion that the hidden layer gives a high-level representation of the inputs. That is, we should be able to reconstruct all the important aspects of the input from the hidden layer output . A natural test is to reconstruct the input itself: the output will be $\hat{\mathbf{x}}_n$, a prediction of the input $\mathbf{x}_n$; and, the error is the difference between the two. For example, using squared error,

$$\mathrm{e}_n = \|\hat{\mathbf{x}}_n - \mathbf{x}_n\|^2.$$

When all is said and done, we obtain the weights without using the targets y_n and the hidden layer gives an encoding of the inputs, hence the name unsupervised auto-encoder. This is reminiscent of the radial basis function network in Chapter 6, where we used an unsupervised technique to learn the centers of the basis functions, which provided a representative set of inputs as the centers. Here, we go one step further and dissect the input-space itself into pieces that are representative of the learning problem. At the end, the targets have to be brought back into the picture (usually in the output layer).

Supervised Deep Network. The previous approach adheres to the philosophical goal that the hidden layers provide an 'intelligent' hierarchical representation of the inputs. A more direct approach is to train the two-layer network on the targets. In this case the output is the predicted target $\hat{y}_n$ and the error measure $\mathrm{e}_n(y_n, \hat{y}_n)$ would be computed in the usual way (for example squared error, cross entropy error, etc.).

In practice, there is no verdict on which method is better, with the unsupervised auto-encoder camp being slightly more crowded than the supervised camp. Try them both and see what works for your problem, that's usually the best way. Once you have your error measure, you just reach into your optimization toolbox and minimize the error using your favorite method (gradient descent, stochastic gradient descent, conjugate gradient descent, ...). A common tactic is to use the unsupervised auto-encoder first to set the weights and then fine tune the whole network using supervised learning. The idea is that the unsupervised pass gets you to the right local minimum of the full network. But, no matter which camp you belong to, you still need to choose the architecture of the deep network (number of hidden layers and their sizes), and there is no magic potion for that. You will need to resort to old tricks like validation, or a deep understanding ☺ of the problem (our hand made network for the '1' versus '5' task suggests a deep network with six hidden nodes in the first hidden layer and two in the second).

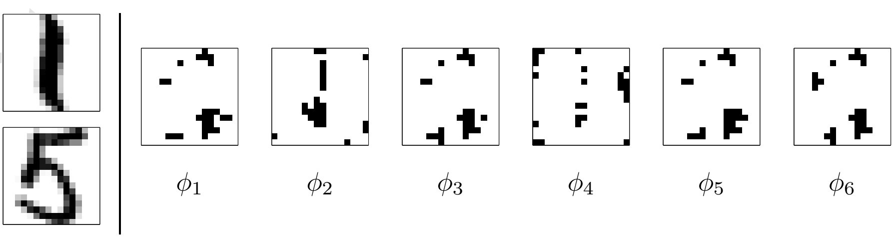
Exercise 7.19

Previously, for our digit problem, we used symmetry and intensity. How do these features relate to deep networks? Do we still need them?

Example 7.5. Deep Learning For Digit Recognition. Let's revisit the digits classification problem '1' versus '5' using a deep network architecture

$$[d^{(0)}, d^{(1)}, d^{(2)}, d^{(3)}] = [256, 6, 2, 1].$$

(The same architecture we constructed by hand earlier, with 16×16 input pixels and 1 output.) We will use gradient descent to train the two layer networks in the greedy algorithm. A convenient matrix form for the gradient of the two layer network is given in Problem 7.7. For the unsupervised auto-encoder the target output is the input matrix X. for the supervised deep network, the target output is just the target vector $\mathbf{y}$. We used the supervised approach with 1,000,000 gradient descent iterations for each supervised greedy step using a sample of 1500 examples from the digits data. Here is a look at what the 6 hidden units in the first hidden layer learned. For each hidden node in the first hidden layer, we show the pixels corresponding to the top 20 incoming weights.



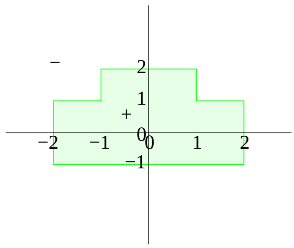
Real data is not as clean as our idealized analysis. Don't be surprised. Nevertheless, we can discern that ϕ_2 has picked out the pixels (shapes) in the typical '1' that are unlikely to be in a typical '5'. The other features seem to focus on the '5' and to some extent match our hand constructed features. Let's not dwell on whether the representation captures human intuition; it does to some extent. The important thing is that this result is automatic and purely data driven (other than our choice of the network architecture); and, what matters is out-of-sample performance. For different architectures, we ran more than 1000 validation experiments selecting 500 random training points each time and the remaining data as a test set.

Deep Network Architecture	E_{in}	E_{test}
$[256, 3, 2, 1]$	0	0.170%
$[256, 6, 2, 1]$	0	0.187%
$[256, 12, 2, 1]$	0	0.187%
$[256, 24, 2, 1]$	0	0.183%

E_{in} is always zero because there are so many parameters, even with just 3 hidden units in the first hidden layer. This smells of overfitting. But, the test performance is impressive at 99.8% accuracy, which is all we care about. Our hand constructed features of symmetry and intensity were good, but not quite this good. □

7.7 Problems

Problem 7.1 Implement the decision function below using a 3-layer perceptron.



Problem 7.2 A set of M hyperplanes will generally divide the space into some number of regions. Every point in $\mathbb{R}^d$ can be labeled with an M dimensional vector that determines which side of each plane it is on. Thus, for example if $M = 3$, then a point with a vector $(-1, +1, +1)$ is on the -1 side of the first hyperplane, and on the +1 side of the second and third hyperplanes. A region is defined as the set of points with the *same* label.

(a) Prove that the regions with the same label are convex.

(b) Prove that M hyperplanes can create at most 2^M distinct regions.

(c) **[hard]** What is the maximum number of regions created by M hyperplanes in d dimensions?

[Answer: $\sum_{i=0}^{d} \binom{M}{d}$ *.]*

[Hint: Use induction and let $B(M, d)$ be the number of regions created by M $(d-1)$-dimensional hyperplanes in d-space. Now consider adding the $(M+1)$th hyperplane. Show that this hyperplane intersects at most $B(M, d-1)$ of the $B(M, d)$ regions. For each region it intersects, it adds exactly one region, and so $B(M+1, d) \le B(M, d) + B(M, d-1)$. (Is this recurrence familiar?) Evaluate the boundary conditions: $B(M, 1)$ and $B(1, d)$, and proceed from there. To see that the $M+1$th hyperplane only intersects $B(M, d-1)$ regions, argue as follows. Treat the $M+1$th hyperplane as a $(d-1)$-dimensional space, and project the initial M hyperplanes into this space to get M hyperplanes in a $(d-1)$-dimensional space. These M hyperplanes can create at most $B(M, d-1)$ regions in this space. Argue that this means that the $M+1$th hyperplane is only intersecting at most $B(M, d-1)$ of the regions created by the M hyperplanes in d-space.]

Problem 7.3 Suppose that a target function f (for classification) is represented by a number of hyperplanes, where the different regions defined by the hyperplanes (see Problem 7.2) could be either classified $+1$ or -1, as with the 2-dimensional examples we considered in the text. Let the hyperplanes be $h_1, h_2, \ldots, h_M$, where $h_m(\mathbf{x}) = \text{sign}(\mathbf{w}_m \bullet \mathbf{x})$. Consider all the regions that are classified $+1$, and let one such region be r^+. Let $\mathbf{c} = (c_1, c_2, \ldots, c_M)$ be the label of any point in the region (all points in a given region have the same label); the label $c_m = \pm 1$ tells which side of h_m the point is on. Define the AND-term corresponding to region r by

$$t_r = h_1^{c_1} h_2^{c_2} \ldots h_M^{c_M}, \text{ where } h_m^{c_m} = \begin{cases} h_m & \text{if } c_m = +1, \\ \bar{h}_m & \text{if } c_m = -1. \end{cases}$$

Show that $f = t_{r_1} + t_{r_2} + \cdots + t_{r_k}$, where $r_1, \ldots, r_k$ are all the positive regions. (We use multiplication for the AND and addition for the OR operators.)

Problem 7.4 Referring to Problem 7.3, any target function which can be decomposed into hyperplanes $h_1, \ldots, h_M$ can be represented by $f = t_{r_1} + t_{r_2} + \cdots + t_{r_k}$, where there are k positive regions.

What is the structure of the 3-layer perceptron (number of hidden units in each layer) that will implement this function, proving the following theorem:

Theorem. Any decision function whose ± 1 regions are defined in terms of the regions created by a set of hyperplanes can be implemented by a 3-layer perceptron.

Problem 7.5 **[Hard]** State and prove a version of a *Universal Approximation Theorem*:

Theorem. Any target function f (for classification) defined on $[0, 1]^d$, whose classification boundary surfaces are smooth, can arbitrarily closely be approximated by a 3-layer perceptron.

[Hint: Decompose the unit hypercube into ϵ-hypercubes ($\frac{1}{\epsilon^d}$ of them); The volume of these ϵ-hypercubes which intersects the classification boundaries must tend to zero (why? – use smoothness). Thus, the function which takes on the value of f on any ϵ-hypercube that does not intersect the boundary and an arbitrary value on these boundary ϵ-hypercubes will approximate f arbitrarily closely, as $\epsilon \to 0$.]

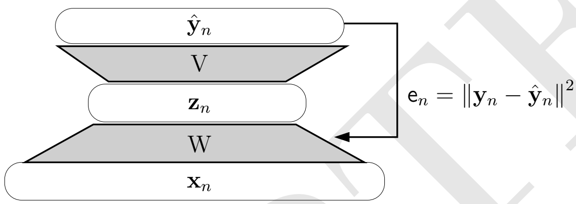
Problem 7.6 The finite difference approximation to obtaining the gradient is based on the following formula from calculus:

$$\frac{\partial h}{\partial w_{ij}^{(\ell)}} = \frac{h(w_{ij}^{(\ell)} + \epsilon) - h(w_{ij}^{(\ell)} - \epsilon)}{2\epsilon} + O(\epsilon^2),$$

where $h(w_{ij}^{(\ell)} + \epsilon)$ to denotes the function value when all weights are held at their values in $\mathbf{w}$ except for the weight $w_{ij}^{(\ell)}$, which is perturbed by ϵ. To get the gradient, we need the partial derivative with respect to each weight.

Show that the computational complexity of obtaining all these partial derivatives $O(W^2)$. *[Hint: you have to do two forward propagations for each weight.]*

Problem 7.7 Consider the 2-layer network below, with output *vector* $\hat{\mathbf{y}}$. This is the two layer network used for the greedy deep network algorithm.



Collect the input vectors $\mathbf{x}_n$ (together with a column of ones) as rows in the input data matrix X, and similarly form Z from $\mathbf{z}_n$. The target matrices Y and $\hat{\mathrm{Y}}$ are formed from $\mathbf{y}_n$ and $\hat{\mathbf{y}}_n$ respectively. Assume a linear output node and the hidden layer activation is $\theta(\cdot)$.

(a) Show that the in-sample error is

$$E_{\text{in}} = \tfrac{1}{N}\text{trace}\left((\mathrm{Y} - \hat{\mathrm{Y}})(\mathrm{Y} - \hat{\mathrm{Y}})^{\mathrm{T}}\right),$$

where

$$\begin{aligned}\hat{\mathrm{Y}} &= \mathrm{ZV}\\ \mathrm{Z} &= [\mathbf{1}, \theta(\mathrm{XW})]\end{aligned} \quad \text{and} \quad \begin{array}{rcl} \mathrm{X} & \text{is} & N \times (d+1)\\ \mathrm{W} & \text{is} & (d+1)\times d^{(1)}\\ \mathrm{Z} & \text{is} & N \times (d^{(1)}+1)\\ \mathrm{V} = \left[\begin{smallmatrix}\mathrm{V}_0\\ \mathrm{V}_1\end{smallmatrix}\right] & \text{is} & (d^{(1)}+1)\times \mathsf{dim}(\mathbf{y})\\ \mathrm{Y}, \hat{\mathrm{Y}} & \text{are} & N \times \mathsf{dim}(\mathbf{y})\end{array}$$

(It is convenient to decompose V into its first row V_0 corresponding to the biases and its remaining rows V_1; $\mathbf{1}$ is the $N \times 1$ vector of ones.)

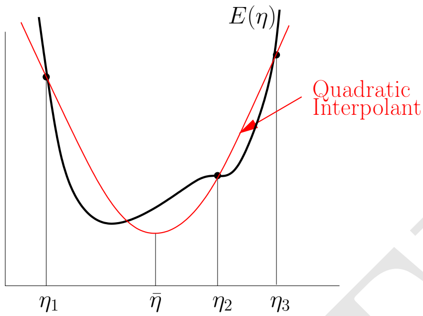
(b) derive the gradient matrices:

$$\begin{aligned}\frac{\partial E_{\text{in}}}{\partial \mathrm{V}} &= 2\mathrm{Z}^{\mathrm{T}}\mathrm{ZV} - 2\mathrm{Z}^{\mathrm{T}}\mathrm{Y}\\ \frac{\partial E_{\text{in}}}{\partial \mathrm{W}}; &= 2\mathrm{X}^{\mathrm{T}}\left[\theta'(\mathrm{XW}) \otimes (\theta(\mathrm{XW})\mathrm{V}_1\mathrm{V}_1^{\mathrm{T}} + \mathbf{1}\mathrm{V}_0\mathrm{V}_1^{\mathrm{T}} - \mathrm{YV}_1^{\mathrm{T}})\right],\end{aligned}$$

where $\otimes$ denotes element-wise multiplication. Some of the matrix derivatives of functions involving the trace from the appendix may be useful.

Problem 7.8 Quadratic Interpolation for Line Search

Assume that a U-arrangement has been found, as illustrated below.



Instead of using bisection to construct the point $\bar{\eta}$, quadratic interpolation fits a quadratic curve $E(\eta) = a\eta^2 + b\eta + c$ to the three points and uses the minimum of this quadratic interpolation as $\bar{\eta}$.

(a) Show that the minimum of the quadratic interpolant for a U-arrangement is within the interval $[\eta_1, \eta_3]$.

(b) Let $e_1 = E(\eta_1)$, $e_2 = E(\eta_2)$, $e_3 = E(\eta_3)$. Obtain the quadratic function that interpolates the three points $\{(\eta_1, e_1), (\eta_2, e_2), (\eta_3, e_3)\}$. Show that the minimum of this quadratic interpolant is given by:

$$\bar{\eta} = \frac{1}{2}\left[\frac{(e_1 - e_2)(\eta_1^2 - \eta_3^2) - (e_1 - e_3)(\eta_1^2 - \eta_2^2)}{(e_1 - e_2)(\eta_1 - \eta_3) - (e_1 - e_3)(\eta_1 - \eta_2)}\right]$$

[Hint: $e_1 = a\eta_1^2 + b\eta_1 + c$, $e_2 = a\eta_2^2 + b\eta_2 + c$, $e_3 = a\eta_3^2 + b\eta_3 + c$. Solve for a, b, c and the minimum of the quadratic is given by $\bar{\eta} = -b/2a$.]

(c) Depending on whether $E(\bar{\eta})$ is less than $E(\eta_2)$, and on whether $\bar{\eta}$ is to the left or right of η_2, there are 4 cases.

In each case, what is the smaller U-arrangement?

(d) What if $\bar{\eta} = \eta_2$, a degenerate case?

Note: in general the quadratic interpolations converge very rapidly to a locally optimal η. In practice, 4 iterations are more than sufficient.

Problem 7.9 [Convergence of Monte-Carlo Minimization]

Suppose the global minimum $\mathbf{w}^*$ is in the unit cube and the error surface is quadratic near $\mathbf{w}^*$. So, near $\mathbf{w}^*$,

$$E(\mathbf{w}) = E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T \mathrm{H}(\mathbf{w} - \mathbf{w}^*)$$

where the Hessian H is a positive definite and symmetric.

(a) If you uniformly sample $\mathbf{w}$ in the unit cube, show that

$$P[E \leq E(\mathbf{w}^*) + \epsilon] = \int\limits_{\mathbf{x}^T \mathrm{H}\mathbf{x} \leq 2\epsilon} d^d\mathbf{x} = \frac{S_d(2\epsilon)}{\sqrt{\det \mathrm{H}}},$$

where $S_d(r)$ is the volume of the $d-$dimensional sphere of radius r,

$$S_d(r) = \pi^{d/2} r^d / \Gamma(\tfrac{d}{2} + 1).$$

[Hints: $P[E \leq E(\mathbf{w}^*) + \epsilon] = P\left[\frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T \mathrm{H}(\mathbf{w} - \mathbf{w}^*) \leq \epsilon\right]$*. Suppose the orthogonal matrix* A *diagonalizes* H*:* $\mathrm{A}^{\mathrm{T}}\mathrm{H}\mathbf{A} = \textit{diag}[\lambda_1^2, \ldots, \lambda_d^2]$*. Change variables to* $\mathbf{u} = \mathrm{A}^{\mathrm{T}}\mathbf{x}$ *and use* $\det \mathrm{H} = \lambda_1^2 \lambda_2^2 \cdots \lambda_d^2$*.]*

(b) Suppose you sample M times and choose the weights with minimum error, $\mathbf{w}_{\min}$. Show that

$$P[E(\mathbf{w}_{\min}) > E(\mathbf{w}^*) + \epsilon] \approx \left(1 - \frac{1}{\pi d}\left(\mu \frac{\epsilon}{\sqrt{d}}\right)^d\right)^N,$$

where $\mu \approx \sqrt{8e\pi/\bar{\lambda}}$ and $\bar{\lambda}$ is the geometric mean of the eigenvalues of H. (You may use $\Gamma(x+1) \approx x^x e^{-x}\sqrt{2\pi x}$.)

(c) Show that if $N \sim (\frac{\sqrt{d}}{\mu\epsilon})^d \log \frac{1}{\eta}$, then with probability at least $1 - \eta$, $E(\mathbf{w}_{\min}) \leq E(\mathbf{w}^*) + \epsilon$.
(You may use $\log(1 - a) \approx -a$ for small a and $(\pi d)^{1/d} \approx 1$.)

Problem 7.10 For a neural network with at least 1 hidden layer and $\tanh(\cdot)$ transformations in each non-input node, what is the gradient (with respect to the weights) if all the weights are set to zero.

Is it a good idea to initialize the weights to zero?

Problem 7.11 **[Optimal Learning Rate]** Suppose that we are in the vicinity of a local minimum, $\mathbf{w}^*$, of the error surface, or that the error surface is quadratic. The expression for the error function is then given by

$$E(\mathbf{w}_t) = E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w}_t - \mathbf{w}^*)\mathrm{H}(\mathbf{w}_t - \mathbf{w}^*) \tag{7.8}$$

from which it is easy to see that the gradient is given by $\mathbf{g}_t = \mathrm{H}(\mathbf{w}_t - \mathbf{w}^*)$. The weight updates are then given by $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta\mathrm{H}(\mathbf{w}_t - \mathbf{w}^*)$, and subtracting $\mathbf{w}^*$ from both sides, we see that

$$\epsilon_{t+1} = (\mathrm{I} - \eta\mathrm{H})\epsilon_t \tag{7.9}$$

Since H is symmetric, one can form an orthonormal basis with its eigenvectors. Projecting ϵ_t and ϵ_{t+1} onto this basis, we see that in this basis, each component

decouples from the others, and letting $\epsilon(\alpha)$ be the αth component in this basis, we see that

$$\epsilon_{t+1}(\alpha) = (1 - \eta\lambda_\alpha)\epsilon_t(\alpha) \tag{7.10}$$

so we see that each component exhibits linear convergence with its own coefficient of convergence $k_\alpha = 1 - \eta\lambda_\alpha$. The worst component will dominate the convergence so we are interested in choosing η so that the k_α with largest magnitude is minimized. Since H is positive definite, all the λ_α's are positive, so it is easy to see that one should choose η so that $1 - \eta\lambda_{min} = 1 - \Delta$ and $1 - \eta\lambda_{max} = 1 + \Delta$, or one should choose. Solving for the optimal η, one finds that

$$\eta_{opt} = \frac{2}{\lambda_{min} + \lambda_{max}} \qquad k_{opt} = \frac{1 - c}{1 + c} \tag{7.11}$$

where $c = \lambda_{min}/\lambda_{max}$ is the condition number of H, and is an important measure of the stability of H. When $c \approx 0$, one usually says that H is ill-conditioned. Among other things, this affects the one's ability to numerically compute the inverse of H.

Problem 7.12 **[Hard]** With a variable learning rate, suppose that $\eta_t \to 0$ satisfying $\sum_t 1/\eta_t = \infty$ and $\sum_t 1/\eta_t^2 < \infty$, for example one could choose $\eta_t = 1/(t+1)$. Show that gradient descent will converge to a local minimum.

Problem 7.13 [Finite Difference Approximation to Hessian]

(a) Consider the function $E(w_1, w_2)$. Show that the finite difference approximation to the second order partial derivatives are given by

$$\frac{\partial^2 E}{\partial w_1^2} = \frac{E(w_1+2h,w_2)+E(w_1-2h,w_2)-2E(w_1,w_2)}{4h^2}$$

$$\frac{\partial^2 E}{\partial w_2^2} = \frac{E(w_1,w_2+2h)+E(w_1,w_2-2h)-2E(w_1,w_2)}{4h^2}$$

$$\frac{\partial^2 E}{\partial w_2^2} = \frac{E(w_1+h,w_2+h)+E(w_1-h,w_2-h)-E(w_1+h,w_2-h)-E(w_1-h,w_2+h)}{4h^2}$$

(b) Give an algorithm to compute the finite difference approximation to the Hessian matrix for $E_{\text{in}}(\mathbf{w})$, the in-sample error for a multilayer neural network with weights $\mathbf{w} = [\mathrm{W}^{(1)}, \ldots, \mathrm{W}^{(\ell)}]$.

(c) Compute the asymptotic running time of your algorithm in terms of the number of weights in your network and then number of data points.

Problem 7.14 Suppose we take a fixed step in some direction, we ask what the optimal direction for this fixed step assuming that the quadratic model for the error surface is accurate:

$$E_{\text{in}}(\mathbf{w}_t + \delta\mathbf{w}) = E_{\text{in}}(\mathbf{w}_t) + \mathbf{g}_t^{\mathrm{T}}\Delta\mathbf{w} + \frac{1}{2}\Delta\mathbf{w}^{\mathrm{T}}\mathrm{H}_t\Delta\mathbf{w}.$$

So we want to minimize $E_{\text{in}}(\Delta\mathbf{w})$ with respect to $\Delta\mathbf{w}$ subject to the constraint that the step size is η, i.e., that $\Delta\mathbf{w}^{\mathrm{T}}\Delta\mathbf{w} = \eta^2$.

(a) Show that the Lagrangian for this constrained minimization problem is:

$$\mathcal{L} = E_{\text{in}}(\mathbf{w}_t) + \mathbf{g}_t^{\mathrm{T}}\Delta\mathbf{w} + \frac{1}{2}\Delta\mathbf{w}^{\mathrm{T}}(\mathrm{H}_t + 2\alpha\mathrm{I})\Delta\mathbf{w}, \tag{7.12}$$

where α is the Lagrange multiplier.

(b) Solve for $\Delta\mathbf{w}$ and α and show that they satisfy the two equations:

$$\begin{aligned}\Delta\mathbf{w} &= -(\mathrm{H}_t + 2\alpha\mathrm{I})^{-1}\mathbf{g}_t,\\ \Delta\mathbf{w}^{\mathrm{T}}\Delta\mathbf{w} &= \eta^2.\end{aligned}$$

(c) Show that α satisfies the implicit equation:

$$\alpha = -\frac{1}{2\eta^2}(\Delta\mathbf{w}^{\mathrm{T}}\mathbf{g}_t + \Delta\mathbf{w}^{\mathrm{T}}\mathrm{H}_t\Delta\mathbf{w}).$$

Argue that the second term is $\theta(1)$ and the first is $O(\sim\|\mathbf{g}_t\|/\eta)$. So, α is large for a small step size η.

(d) Assume that α is large. Show that, To leading order in $\frac{1}{\eta}$

$$\alpha = \frac{\|\mathbf{g}_t\|}{2\eta}.$$

Therefore α is large, consistent with expanding $\Delta\mathbf{w}$ to leading order in $\frac{1}{\alpha}$. *[Hint: expand $\Delta\mathbf{w}$ to leading order in $\frac{1}{\alpha}$.]*

(e) Using (d), show that $\Delta\mathbf{w} = -\left(\mathrm{H}_t + \frac{\|\mathbf{g}_t\|}{\eta}\mathrm{I}\right)^{-1}\mathbf{g}_t$.

Problem 7.15 The outer-product Hessian approximation is $\mathrm{H} = \sum_{n=1}^{N}\mathbf{g}_n\mathbf{g}_n^{\mathrm{T}}$. Let $\mathrm{H}_k = \sum_{n=1}^{k}\mathbf{g}_n\mathbf{g}_n^{\mathrm{T}}$ be the partial sum to k, and let H_k^{-1} be its inverse.

(a) Show that $\mathrm{H}_{k+1}^{-1} = \mathrm{H}_k^{-1} - \dfrac{\mathrm{H}_k^{-1}\mathbf{g}_{k+1}\mathbf{g}_{k+1}^{\mathrm{T}}\mathrm{H}_k^{-1}}{1 + \mathbf{g}_{k+1}^{\mathrm{T}}\mathrm{H}_k^{-1}\mathbf{g}_{k+1}}$.

[Hints: $\mathrm{H}_{k+1} = \mathrm{H}_k + \mathbf{g}_{k+1}\mathbf{g}_{k+1}^{\mathrm{T}}$; and, $(\mathrm{A}+\mathbf{z}\mathbf{z}^{\mathrm{T}})^{-1} = \mathrm{A}^{-1} - \frac{\mathrm{A}^{-1}\mathbf{z}\mathbf{z}^{\mathrm{T}}\mathrm{A}^{-1}}{1+\mathbf{z}^{\mathrm{T}}\mathrm{A}^{-1}\mathbf{z}}$.]

(b) Use part (a) to give an $O(NW^2)$ algorithm to compute H_t^{-1}, the same time it takes to compute H. (W is the number of dimensions in $\mathbf{g}$).

Note: typically, this algorithm is initialized with $\mathrm{H}_0 = \epsilon\mathrm{I}$ for some small ϵ. So the algorithm actually computes $(\mathrm{H} + \epsilon\mathrm{I})^{-1}$; the results are not very sensitive to the choice of ϵ, as long as ϵ is small.

Problem 7.16 In the text, we computed an upper bound on the VC-dimension of the 2-layer perceptron is $d_{\text{vc}} = O(md\log(md))$ where m is the number of hidden units in the hidden layer. Prove that this bound is essentially tight by showing that $d_{\text{vc}} = \Omega(md)$. To do this, show that it is possible to find md points that can be shattered when m is even as follows.

Consider any set of N points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ in general position with $N = md$. N points in d dimensions are in general position if no subset of $d+1$ points lies on a $d-1$ dimensional hyperplane. Now, consider any dichotomy on these points with r of the points classified $+1$. Without loss of generality, relabel the points so that $\mathbf{x}_1, \ldots, \mathbf{x}_r$ are $+1$.

(a) Show that without loss of generality, you can assume that $r \le N/2$. For the rest of the problem you may therefore assume that $r \le N/2$.

(b) Partition these r positive points into groups of size d. The last group may have fewer than d points. Show that the number of groups is at most $\frac{N}{2}$. Label these groups $\mathcal{D}_i$ for $i = 1 \ldots q \le N/2$.

(c) Show that for any subset of k points with $k \le d$, there is a hyperplane containing those points and no others.

(d) By the previous part, let $\mathbf{w}_i, b_i$ be the hyperplane through the points in group $\mathcal{D}_i$, and containing no others. So

$$\mathbf{w}_i^{\mathrm{T}}\mathbf{x}_n + b_i = 0$$

if and only if $\mathbf{x}_n \in \mathcal{D}_i$. Show that it is possible to find h small enough so that for $\mathbf{x}_n \in \mathcal{D}_i$,

$$|\mathbf{w}_i^{\mathrm{T}}\mathbf{x}_n + b_i| < h,$$

and for $\mathbf{x}_n \notin \mathcal{D}_i$

$$|\mathbf{w}_i^{\mathrm{T}}\mathbf{x}_n + b_i| > h.$$

(e) Show that for $\mathbf{x}_n \in \mathcal{D}_i$,

$$\text{sign}(\mathbf{w}_i^{\mathrm{T}}\mathbf{x}_n + b_i + h) + \text{sign}(-\mathbf{w}_i^{\mathrm{T}}\mathbf{x}_n - b_i + h) = 2,$$

and for $\mathbf{x}_n \notin \mathcal{D}_i$

$$\text{sign}(\mathbf{w}_i^{\mathrm{T}}\mathbf{x}_n + b_i + h) + \text{sign}(-\mathbf{w}_i^{\mathrm{T}}\mathbf{x}_n - b_i + h) = 0$$

(f) Use the results so far to construct a 2-layer MLP with $2r$ hidden units which implements the dichotomy (which was arbitrary). Complete the argument to show that $d_{\text{vc}} \ge md$.